Derrière le consensus : coordination faiblement contrainte dans les systèmes distribués asynchrones

Corentin Travers

N° d'ordre: 3661

THÈSE

Présentée devant

devant l'Université de Rennes 1

pour obtenir

le grade de : Docteur de l'Université de Rennes 1
Mention Informatique

par

Corentin Travers

Équipe d'accueil : asap
École Doctorale : Matisse
Composante universitaire : IFSIC/IRISA

Titre de la thèse :

Derrière le consensus :
Coordination faiblement contrainte dans
les sytèmes distribués asynchrones

soutenue le 20 novembre 2007 devant la commission d'examen

M. :	Patrice	Quinton	Président
MM. :	Hagit	Attiya	Rapporteurs
	Rachid	Guerraoui	
MM. :	Achour	Mostéfaoui	Examinateurs
	Nir	Shavit	
	Michel	Raynal	

Le directeur des travaux, Prof. Michel Raynal

Table des matières

Introduction

Un *système distribué* est une collection de calculateurs qui communiquent entre eux. Par exemple, les architectures multi-processeurs modernes, les réseaux locaux de stations de travail ou Internet sont des systèmes distribués. Un système distribué ou *réparti* est donc constitué de deux types d'entités, les calculateurs ou *processus* et un système (appelé aussi *médium*) de communication. Du fait de leur nature répartie, ces systèmes sont sujets à différents types d'incertitude qui rendent non triviale, voire impossible, la résolution de nombreux problèmes simples.

Incertitudes Ces incertitudes peuvent grossièrement être classées en trois catégories :

1. *Incertitude temporelle.* À la différence d'un système centralisé, il n'existe pas d'horloge globale à laquelle les processus peuvent se référer. D'autre part, les délais d'exécution d'une même action par différents processus sont sujets à de grandes variations et sont souvent imprévisibles. En particulier, les temps de transfert de données entre les calculateurs sont sujets à de grandes variations. Ces délais varient par exemple en fonction de la charge locale de la machine sur laquelle un processus s'exécute ou dépendent de l'état du réseau. L'estimation de ces délais met donc en jeu un grand nombre de paramètres et l'on préfère souvent faire l'hypothèse que les systèmes sont purement *asynchrones*. Les processus ont connaissance de leur horloge locale, mais n'ont pas accès à une horloge globale : les horloges locales ne sont pas synchronisées par un périphérique extérieur.
2. *Incertitude sur l'état du système.* Il n'est pas réaliste de supposer que le système ne connaîtra pas de défaillances. Plus le système est grand, plus le risque que des composants matériels ou logiciels soient défaillants est important. Les défaillances touchent les processus (panne franche, comportement byzantin, par exemple) ainsi que le médium de communication (perte de messages, corruption des données transitant sur le réseau, etc.).
3. *Incertitude structurelle.* Ces dernières années, on assiste à une large augmentation du nombre d'entités interconnectées au sein d'un même système (par exemple, réseau de téléphonie mobile, système pair à pair, etc.). Dans de tels systèmes, les utilisateurs se connectent par intermittence et la structure du médium de communication est susceptible de changer au cours du temps du fait par exemple de la mobilité des usagers. Enfin, le nombre d'entités qui composent le système à chaque instant est très grand. Il n'est pas raisonnable de supposer une connaissance

locale de l'ensemble du système.

Dans cette thèse, nous nous intéressons principalement aux incertitudes de type 1 et 2.

Besoin de coordination La mise en œuvre d'un service au travers d'un système distribué nécessite un certain degré de *coordination* entre les sites de calcul. Les sites distants effectuent des calculs locaux et communiquent entre eux dans le but de réaliser une tâche commune. L'accomplissement du but commun nécessite la prise de décisions communes. Par exemple, les processus doivent régulièrement se mettre d'accord sur l'état d'avancement du calcul global, ou se synchroniser pour régler les accès concurrents à des structures de donnée partagées.

Pour tenter de mettre en lumière ce besoin de coordination et la difficulté de sa mise en œuvre, considérons l'exemple suivant. Dans le système, il existe un certain nombre de ressources $r_1, \ldots, r_\alpha$. Pour mener à bien son calcul, chaque processus doit accéder à l'une de ces ressources en accès exclusif. Un processus ne cherche pas à acquérir une ressource particulière. Il s'agit simplement de garantir qu'à chaque instant, au plus un processus utilise une ressource donnée. Par exemple, le système est composé d'avions en approche d'un aéroport. Pour éviter les collisions, chaque avion doit choisir un couloir aérien exclusif. Ou, plus prosaïquement, il s'agit d'attribuer des places de parking à un ensemble de véhicules.

Une façon simple de résoudre ce problème est de désigner une entité particulière, le coordonnateur, qui sera chargé d'attribuer les ressources aux processus demandeurs. C'est le rôle de la tour contrôle de l'aéroport. Cependant, dans cette *architecture centralisée* le degré de *tolérance aux défaillances* est faible puisque la panne de l'autorité centrale entraîne l'écroulement complet du système.

Pour éviter cela, une idée naturelle consiste à répliquer le coordonnateur : plusieurs machines assurent la fonction coordinatrice dans le but de pallier à l'arrêt inopiné du coordonnateur. Dans la *réplication passive*, le coordonnateur sauvegarde régulièrement son état courant (quelles sont les ressources utilisées et par qui) sur un ensemble de *réplicas*. S'il tombe en panne, l'un des réplicas est choisi pour prendre le relais. Il reprend l'exécution à partir du dernier état sauvegardé. Cette technique nécessite donc la définition de points de reprise cohérents. Remarquons également qu'une partie du travail effectué peut être perdue. En effet, il est possible que les dernières actions effectuées par le coordonnateur avant de défaillir n'apparaissent pas dans le dernier point de reprise sauvegardé. Ainsi, cette solution est inadaptée à la gestion du trafic aérien.

La *réplication active* vise à tolérer les défaillances de façon transparente. Chacun des réplicas, au lieu de stocker passivement les états successifs du coordonnateur se comporte comme le coordonnateur. Pour maintenir la cohérence de l'allocation des ressources, il est nécessaire de maintenir une coordination forte entre les réplicas. Pour s'en apercevoir, envisageons quel pourrait être le traitement d'une demande d'attribution de couloir par un réplica m. m ne peut unilatéralement attribuer un couloir r_i au demandeur. En effet, un autre réplica pourrait attribuer en même temps la même ressource r_i à un autre demandeur. Ce cas de figure se produit si, du fait de délais de transmission non uniformes, les demandes n'arrivent pas dans le même ordre sur chacun des réplicas. Les

réplicas doivent donc se mettre d'accord sur l'attribution des ressources aux processus ou sur l'ordre de traitement des demandes.

Enfin, une dernière solution consiste à se passer complètement de coordinateur. Les demandeurs effectuent un algorithme *décentralisé* pour gérer l'attribution des ressources. L'existence d'un tel algorithme n'est a priori pas évidente. En effet, dans le cas le plus défavorable, les demandeurs sont susceptibles de tomber en panne inopinément et les délais de communication sont imprévisibles. Les solutions du type « attendre la permission de l'ensemble des autres demandeurs avant d'acquérir une ressource » sont donc vouées à l'échec. Il s'avère que l'existence d'une solution décentralisée est conditionnée par le rapport du nombre de ressources disponibles au nombre de demandeurs. Par exemple, si l'on suppose qu'à tout moment au plus 3 avions sont en approche de l'aéroport, il suffit de prévoir 7 couloirs aériens. Il a également été montré que ce nombre est nécessaire. Autrement dit, il n'existe pas d'algorithme décentralisé qui résout ce problème pour 3 processus susceptibles de subir des défaillances dans un environnement *asynchrone* si le nombre de ressources est inférieur à 7.

Maîtriser l'incertitude Nous voyons à travers cet exemple se dessiner le besoin de se *mettre d'accord*. Un problème élémentaire, brique fondamentale de la mise en œuvre de services distribués tolérant les défaillances est appelé le *consensus*. Sa spécification est très simple. Les processus proposent initialement une valeur et doivent s'accorder sur une valeur commune choisie parmi les propositions initiales. Malheureusement, l'un des résultats fondamentaux du calcul distribué [48] est négatif : il n'existe pas de solution déterministe qui tolère les pannes lorsque le système est totalement asynchrone.

Plus généralement, un pan de la théorie du calcul distribué est consacré à l'étude de la calculabilité : quels sont les problèmes qu'il est possible de résoudre en environnement distribué. C'est une question très vaste. Il existe en effet une grande variété de modèles de systèmes distribués qui reflètent différentes hypothèses sur l'architecture du réseau, le médium de communication ou le comportement des processus défaillants. De plus, changer l'un des paramètres du modèle peut entraîner une diminution drastique de la classe des problèmes qu'il est possible de résoudre. Par exemple, si le médium de communication garantit un délai maximal sur le temps de transfert des messages, l'impossibilité du consensus ne tient plus.

Un aspect marquant de cette théorie est le nombre considérable de résultats négatifs. Dans [47], Fich et Rupper énumèrent plusieurs centaines de résultats d'impossibilité pour différents modèles. En quoi ces résultats sont importants pour le calcul distribué ? Ils nous aident à comprendre l'essence du calcul distribué : pour quelles raisons certains problèmes sont difficiles, quelle caractéristique rend un modèle puissant et comment des modèles différents se comparent. Ils indiquent quelles sont les approches qui n'aboutiront pas lorsque l'on cherche une solution à un problème donné. Si le problème doit être résolu, la preuve de son impossibilité suggère des directions pour modifier légèrement sa spécification ou comment renforcer le modèle pour obtenir une solution raisonnable. Enfin, tenter d'établir une impossibilité peut mener à la découverte de nouveaux algorithmes.

Coordination faible en environnement asynchrone Cette thèse contribue à la théorie du calcul distribué tolérant les défaillances. Nous nous plaçons dans un environnement asynchrone (ni les délais de transmission des données, ni les vitesses des processus ne sont connus) dans lequel les processus sont susceptibles de s'arrêter inopinément (panne franche ou *crash*). Dans ce contexte, les chercheurs ont formulé plusieurs problèmes simples qui capturent différents schémas de coordination. Nous avons rencontré deux problèmes qui rentrent dans cette catégorie : le consensus qui requiert que les processus s'accordent sur l'une des propositions et l'allocation de ressources. Le schéma d'allocation de ressource esquissé ci-dessus est connu sous le nom du problème du *renommage*. Chaque participant possède initialement un identifiant unique appartenant à un vaste espace et doit sélectionner un nouveau nom unique dans un espace plus petit.

Quelles sont, du point de vue de la calculabilité, les relations entre ces deux problèmes ? Il est clair qu'à partir d'une solution au problème du consensus, il est possible de construire un algorithme pour le renommage - les processus se mettent successivement d'accord sur l'attribution des noms. En fait, il a été montré qu'un *objet* consensus (c'est-à-dire une « boîte » qui résout ce problème) est universel [69]. Dans un système muni d'objets consensus, tout problème qui possède une spécification séquentielle peut être résolu. Au contraire, la coordination offerte par un objet renommage est strictement plus faible : il est en général impossible de résoudre le consensus à l'aide d'objets renommage. Par opposition au consensus, nous dirons que le renommage est un problème de coordination faible. Nous nous intéressons aux relations, du point de vue de la calculabilité, entre différentes incarnations de la coordination « faible ».

Puisque le consensus est, en un certain sens, universel, il est naturel de classer les problèmes ou les objets par rapport au consensus. Dans la *hiérarchie du consensus* [69], le rang[1] d'un objet O est le plus grand nombre de processus pour lesquels il existe un algorithme, solution au problème du consensus, qui utilise des objets O. En particulier, un objet de rang 1 n'est d'aucune utilité pour résoudre le problème du consensus. Un mot mémoire atomique a pour rang 1. Quel est le rang du renommage dans cette hiérarchie ? La réponse dépend du cardinal de l'espace des nouveaux noms. Dans le meilleur des cas (taille de l'espace = nombre de participants), le rang du renommage est 2. Dès que l'espace de renommage est supérieur au nombre de participants, le rang s'effondre et devient 1. Plus généralement, nous nous intéressons à la classe des problèmes qui ne sont pas capturés par la hiérarchie du consensus. Le rang d'un objet qui implémente un problème de cette famille est au mieux 2, en fonction des valeurs des paramètres qui le définissent. Le grain de la hiérarchie du consensus n'est pas suffisamment fin pour établir des relations entre ces problèmes. Ainsi, du seul rang du renommage et d'un mot mémoire, il est impossible d'en déduire l'existence ou l'impossibilité de résoudre le renommage à l'aide de mots mémoires.

Le problème emblématique de cette famille, qui a reçu une attention considérable dans la littérature est l'*accord ensembliste*. De même que pour le consensus, chaque processus doit décider une valeur sélectionnée parmi les propositions initiales. Cepen-

[1] La définition exacte est donnée dans [69]. Elle sera également précisée par la suite (paragraphe 1.4.6).

dant, la propriété d'accord est moins contrainte puisque le nombre de valeurs décidées collectivement est borné par k où k est un paramètre du problème. Ce paramètre k caractérise en un certain sens le degré de coordination. Pour $k = 1$, nous retrouvons le problème du consensus.

Parallèlement, des travaux ont cherché à quantifier l'incertitude inhérente au monde distribué en paramétrant les modèles. Ainsi, un paramètre souvent pris en compte est une borne t sur le nombre maximal de pannes susceptibles de se produire. Ce paramètre caractérise dans une certaine mesure le degré d'incertitude du modèle. On comprend alors mieux l'intérêt porté à l'accord ensembliste. Étudier la calculabilité de ce problème, en fonction des paramètres t et k promet d'obtenir une meilleur compréhension de la difficulté de la coordination en environnement distribué. De fait, cette ligne de recherche a été fructueuse. Elle a notamment été à l'origine de l'introduction des techniques issues de la topologie dans le monde du calcul distribué. Ces techniques ont donné naissance à une nouvelle façon d'appréhender le calcul distribué, comme déformation d'espaces topologiques. Les déformations autorisées dépendent de la puissance des objets dont le modèle est équipé, ou du degré d'incertitude (synchrone ou asynchrone, borne sur le nombre de défaillances, etc.). Le principal résultat est le suivant : en environnement asynchrone, il existe une solution à l'accord ensembliste si, et seulement si, $k > t$.

Contributions Cette thèse explore la question suivante : Quel degré de coordination peut être atteint en fonction du degré d'incertitude du systèmes sous-jacent ?

- Le chapitre 2 explore les relations entre différents schémas de coordination : accord ensembliste, renommage et consensus simultanés. Il s'avère que malgré leur nature apparemment différente, il existe une certaine unité entre ces problèmes.
- Le chapitre 3 compare les différentes hypothèses exprimées dans le formalisme des *détecteurs de défaillances* qui permettent de circonvenir l'impossibilité de l'accord ensembliste.
- Enfin, dans le chapitre 4 nous cherchons à unifier l'approche « détecteurs de défaillances » et la vision « topologique » de l'algorithmique distribuée. Les deux approches visent, dans des cadres abstraits différents, à maîtriser l'incertitude dans le but de coordonner des processus au sein de systèmes répartis. Nous défendons l'idée qu'un cadre unifié facilite la conception d'algorithmes fondés sur les détecteurs de défaillances ainsi que la démonstration de résultats d'impossibilités.

Les paragraphes qui suivent développent les points ci-dessus et précisent la démarche adoptée.

L'accord ensembliste n'est qu'une facette des schémas de coordination. Le renommage abstrait un autre schéma de coordination, en apparence de tout autre nature. Nous introduisons également un nouveau schéma, qui affaiblit le consensus mais d'une façon différente. L'idée est de considérer plusieurs instances du consensus en parallèle, chaque processus devant décider dans au moins l'une de ces instances. Là encore, ce problème ne peut être caractérisé finement dans la hiérarchie du consensus (son rang est 1) et nous nous interrogeons sur sa difficulté relative par rapport aux deux problèmes évoqués précédemment. Quelles sont les relations entre ces problèmes ? À partir d'une solution à l'un d'entre eux, est-il possible de résoudre l'autre ? Pour quelles valeurs des

paramètres qui entrent dans la spécification (borne sur le nombre de valeurs décidées, taille de l'espace de renommage ou nombre de consensus simultanés) ? Ces questions sont développées dans le chapitre 2. Nous montrons que malgré leur nature a priori différente, il existe un lien fort entre ces problèmes.

La démarche que nous défendons ici est purement algorithmique. Nous cherchons à procéder par réductions, une démarche classique dans le monde séquentiel. Au lieu d'adapter les démonstrations d'impossibilité existantes ou d'en concevoir de nouvelles, nous cherchons à établir les impossibilités par réduction algorithmique en des résultats connus. On peut dresser un parallèle avec l'étude de la classe des problèmes NP-complets. La première démonstration de l'existence de problèmes NP-complets [38] repose sur l'utilisation de machines de Turing. Par analogie, la démonstration de l'impossibilité de l'accord ensembliste repose sur l'utilisation d'outils mathématiques issus de la topologie algébrique. Par contre, il n'est pas nécessaire d'être un expert dans la manipulation des machines de Turing pour montrer qu'un problème est complet dans la classe NP. On préférera procéder par réduction algorithmique à partir de problèmes similaires dont l'appartenance à cette classe est connue. On souhaiterait qu'il en soit de même dans le monde distribué. De ce point de vue, cette thèse enrichit le « Garey-Johnson » [60] du distribué en introduisant de nouveaux problèmes ainsi que des techniques algorithmiques originales.

Dans une deuxième partie, nous étudions les relations entre les différents modèles dans lesquels il existe des solutions aux schémas de coordination mentionnés précédemment. Un modèle décrit le médium de communication et le comportement des processus en temps normal ainsi qu'en cas de défaillance. Il borne également l'incertitude. Par exemple, il peut spécifier le nombre maximal de processus défaillants (paramètre t), des bornes sur les temps de transfert des messages ou sur la vitesse des processus. Un formalisme puissant pour exprimer ces différentes restrictions est celui des *détecteurs de défaillances* [30]. L'idée est d'enrichir le modèle complètement asynchrone avec des oracles qui fournissent des informations relatives au motif des défaillances de l'exécution courante. Ces oracles sont spécifiés par des propriétés abstraites par oppositions aux propriétés physiques du réseau sous-jacent. La pertinence des informations fournies est supposée abstraire des propriétés de synchronie du système sous-jacent. Par exemple, un oracle détecteur de défaillances indiquera la liste des processus défaillants. Dans un systèmes synchrone (les délais de transmission des messages sont bornés et connus), il est facile de mettre en œuvre un tel détecteur à l'aide d'envois périodiques de messages « je suis vivant » et délais de garde.

Dans la littérature, plusieurs familles de détecteurs ont été introduites dans le but de résoudre l'accord ensembliste. Dans le chapitre 3, nous étudions ces différentes familles et cherchons à les classer. Nous répondons à des questions du type : Étant donnés deux détecteurs $\mathcal{C}1$ et $\mathcal{C}2$, est-il possible de mettre en œuvre $\mathcal{C}1$ à l'aide de $\mathcal{C}2$? Peut-on les combiner pour obtenir un détecteur $\mathcal{C}$ plus puissant ? Plus puissant s'entend ici comme un détecteur à l'aide duquel il est possible de résoudre un problème plus difficile que les problèmes qu'il est possible de résoudre en utilisant uniquement $\mathcal{C}1$ ou $\mathcal{C}2$.

Dans l'approche « détecteurs de défaillances », le modèle asynchrone est enrichi par des oracles. Le modèle enrichi est alors plus puissant : certains problèmes autrement

impossibles à résoudre possèdent maintenant des solutions. Intuitivement, l'adjonction de l'oracle entraîne la diminution de l'incertitude. Cependant, la nature de cette restriction demeure inconnue. D'autre part, l'approche « topologique » a amené la définition de modèles dans lesquels les exécutions sont hautement structurées. Ces modèles facilitent l'examen de l'ensemble des exécutions possibles et donc aident à établir des résultats d'impossibilité. Du fait de leur grande structure, il est plus facile d'obtenir des résultats positifs en construisant des algorithmes. Dans le chapitre 4, nous cherchons à mieux comprendre comment les détecteurs restreignent l'incertitude du système en étendant ces modèles structurés pour prendre en compte les détecteurs de défaillances. Nous pensons également que cette approche pourrait être bénéfique pour la recherche du détecteur nécessaire et suffisant pour résoudre un problème donné.

Chapitre 1

Cadre de l'étude

Dans ce chapitre, nous définissons les modèles de systèmes que nous utiliserons et précisons les notions évoquées dans l'introduction. Nous introduisons d'abord le modèle de calcul dans le paragraphe 1.1. Le paragraphe 1.3 présente l'outillage algorithmique ayant trait à la synchronisation d'informations globales. Pour résoudre un problème donné, il est souvent utile que les processus obtiennent des vues des informations qui « ne diffèrent pas trop ». Les primitives, construites à partir des opérations de base des modèles choisis, permettent l'obtention de telles vues. La paragraphe 1.4 formalise la notion de problème de coordination et présente succinctement les principaux résultats liés à la calculabilité de problèmes d'accord usuels. Enfin, le paragraphe 1.5 résume les principaux résultats obtenus au cours de cette thèse.

1.1 Modèles de systèmes distribués

Notre étude se déroule dans les modèles standards du calcul distribué. Nous renvoyons le lecteur à [121, 82] pour une présentation exhaustive et détaillée.

1.1.1 Processus

On considère un système statique, c'est-à-dire composé d'un nombre fini d'entités de calcul ou *processus*. Les processus sont indexés de 1 à n ($n > 1$). On notera $\Pi = \{p_1, \ldots, p_n\}$ l'ensemble des processus qui composent le système. L'index i du processus p_i n'est pas nécessairement connu de p_i. Cet index simplifie la description des algorithmes et la présentation des démonstrations.

Les processus ne sont pas anonymes. Chaque processus possède une *identité* propre. Par exemple, un processus peut être identifié par le couple (adresse IP, numéro de port) dans un réseau local. L'identité id_i est initialement connue de p_i mais n'est pas nécessairement connu des autres processus. On supposera cependant que les identités sont deux à deux distinctes et totalement ordonnées (par exemple, les identités sont tirées parmi les entiers naturels). Lorsque toutes les identités sont initialement connues, nous identifierons l'index i de p_i avec son identité.

Pour simplifier la présentation du modèle, on suppose qu'il existe une horloge discrète globale et commune à tous les processus. Il s'agit d'un périphérique fictif : les processus n'y ont pas accès. $\mathbb{T}$ désigne l'ensemble des valeurs pouvant être prises par cette horloge. On confond $\mathbb{T}$ avec l'ensemble des entiers naturels.

Défaillance des processus Il est peu réaliste de considérer que tous les composants d'un système sont fiables. Plus on augmente le nombre de machines ou le nombre de programmes, plus la probabilité d'une faute matérielle ou logicielle est importante. Les processus sont susceptibles de connaître des défaillances. Dans cette thèse, nous nous limitons aux défaillances de type *panne franche* ou *crash*. Dans ce modèle de faute, un processus défaillant arrête soudainement d'exécuter le code qui lui a été attribué et n'effectue alors aucun calcul. En particulier, il cesse de communiquer avec les autres processus. Étant donnée une exécution, un processus est dit *correct* s'il ne tombe pas en panne au cours de cette exécution. Dans le cas contraire, il est dit *fautif* ou *défaillant*. Le paramètre t, $(0 < t < n)$ dénote le nombre maximal de processus défaillants dans toute exécution. Dans une exécution donnée, nous noterons $f (0 \leq f \leq t)$ le nombre effectif de processus défaillants.

Motif de défaillances Un *motif de défaillance* F est une fonction de $\mathbb{T}$ vers 2^{Π}, où $F(\tau)$ représente l'ensemble des processus défaillants à l'instant τ. Dès qu'un processus tombe en panne, il ne peut reprendre son activité, d'où l'inclusion $\forall \tau \in \mathbb{N} : F(\tau) \subseteq F(\tau + 1)$ Nous notons $\mathbb{F}$ l'ensemble des motifs de défaillances.

Soit $F \in \mathbb{F}$. L'ensemble des processus fautifs $\mathit{Fautif}(F)$ dans le motif F est défini par $\mathit{Fautif}(F) = \cup_{\tau \in \mathbb{T}} F(\tau)$. De même, l'ensemble des processus corrects $\mathit{Correct}(F)$ est défini par $\mathit{Correct}(F) = \Pi - \mathit{Fautif}(F)$. Un processus $p_i \notin F(\tau)$ est dit *vivant* à l'instant τ et défaillant dans le cas contraire.

La notion d'*environnement* $\mathcal{E}$ caractérise les motifs de défaillance admissibles. Nous considérons principalement l'environnement $\mathcal{E}_t$ qui contient tous les motifs dans lesquels au plus t processus sont défaillants. Formellement, $\mathcal{E}_t = \{F \in \mathbb{F} : |\mathit{Fautif}(F)| \leq t\}$. Pour $F \in \mathcal{E}_t$, le cardinal de l'ensemble des processus fautifs $\mathit{Fautif}(F)$ est noté f $(0 \leq f \leq t)$. On supposera également l'existence d'au moins un processus correct ainsi que la possibilité d'au moins une défaillance, i.e., $1 \leq t \leq n - 1$.

1.1.2 Médium de communication

Une autre caractéristique importante des systèmes distribués est l'interface de communication. On distingue principalement deux modèles de communication : *passage de messages* et *mémoire partagée*. Dans le premier modèle les processus communiquent en s'échangeant des messages qui transitent via un réseau de communication. Dans le second, les processus communiquent en accédant à des objets distribués comme par exemple des registres, des piles, etc.

1.1.2.1 Modèle à messages

Dans ce modèle, les processus communiquent en s'échangeant des messages. Les processus sont reliés par des canaux de communication dont l'interface comporte deux primitives :

- send(msg) envoie le message msg sur le canal considéré.
- receive() retourne le premier message dans la file d'attente du canal.

Nous ne faisons d'hypothèse ni sur l'ordre de réception des messages ni sur les délais de transmission. Cependant, chaque canal est *fiable* : il ne perd, n'altère ni ne crée de nouveaux messages. En particulier, si p_i envoie un message m alors le destinataire le recevra à moins qu'il ne défaille. Nous supposons que la topologie du réseau est le graphe complet. Chaque paire de processus est donc connecté par un canal. Cette hypothèse est assez réaliste car on ne s'intéresse pas à la complexité du routage. Sur un réseau comme Internet, deux processus peuvent ouvrir une connection s'ils le souhaitent.

La primitive de diffusion broadcast(m) est un racourci pour le code suivant : **foreach** $j \in \Pi$ **do** send(m) *to* p_j **enddo**. Si l'émetteur p_i défaille au cours de l'exécution de cette boucle, m n'est peut être pas envoyé à certains processus. Pour pallier ce problème et dans le but de simplifier la conception des algorithmes, nous enrichissons le modèle avec une primitive de *diffusion fiable* [67]. L'abstraction diffusion fiable offre deux primitives R_bcast(m)/R_deliver() qui permettent aux processus de diffuser et recevoir des messages (Nous dirons alors qu'un processus $R\_$ *diffuse* ou $R\_$ *livre* un message m).

Une primitive de diffusion fiable[1] R_bcast()/R_deliver() est définie par les propriétés suivantes :

Définition 1.1 (Diffusion fiable) *La paire de primitives R_bcast()/R_deliver() de la diffusion fiable satisfait (les messages sont supposés uniques) :*

- Validité *Si un processus $R\_$ livre m alors un processus a précédemment $R\_$ diffusé m ;*
- Intégrité *Un processus $R\_$ livre un message m au plus une fois ;*
- Terminaison *Si un processus correct $R\_$ diffuse un message m alors il $R\_$ livre le message m ;*
- Accord *Si un processus $R\_$ livre m alors m est $R\_$ livré par tous les processus corrects.*

La diffusion fiable impose donc que les processus s'accordent sur les messages livrés mais n'impose pas d'ordre sur la livraison des messages. De même, ces primitives n'offrent pas de garantie temporelles. Un message $R_$diffusé sera reçu par un processus correct, cependant le délai de transmission n'est pas connu ni borné a priori. On observe enfin que les processus corrects livrent le même ensemble de messages E. Par contre l'ensemble des messages $R_$livrés par les processus fautifs est un sous-ensemble de E. Il est possible d'implémenter les primitives R_bcast()/R_deliver() à partir des primitives de base send()/receive() en tolérant un nombre quelconque de processus fautifs [67]. On n'augmente donc pas puissance de calcul du modèle en supposant que les processus ont accès aux primitives R_bcast()/R_deliver().

[1] *Reliable broadcast* dans la littérature.

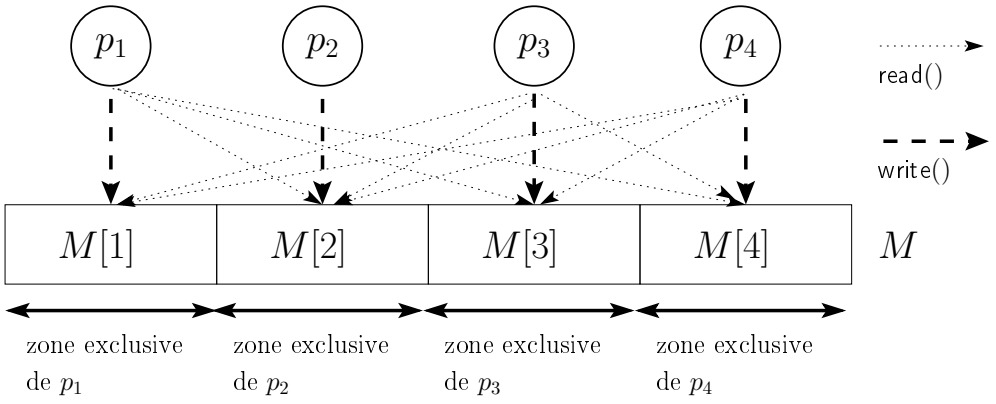

FIG. 1.1 – Mémoire partagée à écrivain unique, $n = 4$

1.1.2.2 Mémoire partagée

Dans un modèle à mémoire partagée, les processus partagent une entité commune, la mémoire. Pour simplifier le raisonnement, nous supposons que cette mémoire est composée en *registres atomiques* communs [69, 80]. Nous considérons dans cette étude des registres à *écrivain unique*. Chaque registre ne peut être écrit que par un processus donné. La mémoire peut être vue comme un tableau M divisé en n zones, chaque zone correspondant à la zone d'écriture d'un processus donné. Un tel système est représenté dans la figure 1.1.

1.1.2.3 Réduction entre modèles

Dans cette thèse, nous nous intéressons principalement à la calculabilité. À ce titre, une question naturelle est la puissance relative des deux modèles exposés ci-dessus. Etant donné un problème P que l'on peut résoudre dans l'un des modèles de communication, peut-on le résoudre dans l'autre modèle de communication ?

Il n'est pas difficile de se convaincre que dans le modèle à mémoire partagé, il est aisé de simuler le modèle à passage de messages. Par exemple, chaque registre $R[i]$ est divisé en n zones. Pour simuler l'envoi d'un message m vers p_j, p_i écrit m dans la i-ième zone de $R[i]$. p_j peut alors obtenir les messages qui lui sont adressés en observant chacun des registres.

Dans l'autre direction, il a été montré comment construire un registre atomique dans le modèle à passage de messages [14]. Cependant, cette simulation requiert une majorité de processus corrects ($t < \frac{n}{2}$).

1.1.3 Aspects temporels

Dans le cas d'un système centralisé, on suppose en général qu'il existe une horloge globale accessible par les processus, qui permet de dater de façon univoque une action vis à vis des actions des autres processus. Dans un système distribué, la notion de temps est locale à chaque processus.

Systèmes synchrones Dans le modèle *synchrone* [36], les processus exécutent des rondes successives. Durant chaque ronde, les processus communiquent entre eux et effectuent des calculs locaux. Lorsqu'un processus termine la ronde r, il a la certitude qu'à cet instant tous les processus corrects qui participent au protocole ont également terminé cette ronde. Ce modèle est très puissant : il est possible de résoudre des problèmes que l'on ne peut pas résoudre dans d'autres modèles (typiquement, le consensus peut être résolu dans ce type de système [82]). La propriété d'un système synchrone est qu'il existe une borne supérieure sur le temps de transmission des messages et une borne maximale sur le temps d'exécution d'une étape de calcul. De plus, les processus connaissent ces bornes et disposent d'horloges synchronisées. La détection des processus défaillants est donc facilement mise en œuvre par un mécanisme de type délais de garde.

Cependant, en dehors de certaines architectures dédiées, il est souvent impossible d'établir des bornes supérieures utilisables sur le temps de transfert des messages ou le temps d'exécution des processus. Néanmoins, dans le cadre plus théorique de la calculabilité, il est intéressant de rechercher des solutions algorithmiques dans ce modèle. Un résultat d'impossibilité s'étendra immédiatement dans tout modèle qui autorise un comportement synchrone. De plus, il existe un lien fort entre efficacité synchrone et degré de tolérance aux fautes dans le modèle asynchrone [50, 101, 91].

Systèmes asynchrones Dans un système *asynchrone*, il n'existe pas de bornes sur les temps de calcul, de transmission des messages ou de lecture/écriture des registres. Ce modèle est simple et correspond aux observations des systèmes réels. En effet, en fonction de la charge du réseau ou des processeurs, les temps de calcul ou les délais de transmissions sont variables et difficiles à prévoir. Bien que plus réaliste que le modèle synchrone, le modèle asynchrone souffre de sa trop grande généralité : il est très faible et de nombreux problèmes n'ont pas de solution dans ce modèle.

Dans le modèle synchrone, l'écoulement du temps offre un repère commun aux processus. Par contre, dans le cas purement asynchrone, seule l'interface de communication peut offrir un tel repère. Plus ce médium offre de garanties, plus les algorithmes seront simples et efficaces. À titre d'exemple, le médium de communication peut assurer un ordre sur la livraison des messages, la persistance des données envoyées ou une synchronisation plus ou moins forte sur les vues successives de ces données. Plus le médium de communication offre de garanties, plus la conception d'algorithmes sera facilitée. Nous verrons dans le paragraphe 1.3 différentes primitives de communication qui offrent plus ou moins de garanties.

Entre synchrone et asynchrone Les modèles synchrones et asynchrones représentent deux extrêmes dans l'ensemble des modèles possibles. Dans le modèle synchrone pur, tous les paramètres temporels sont bornés et ces bornes sont connues. À l'inverse, rien n'est connu sur les paramètres temporels dans les modèles asynchrones purs. Plusieurs modèles dits *partiellement synchrones* ont aussi été proposés afin de se rapprocher du comportement d'un système réel.

Dwork, Lynch et Stockmeyer [45] ont introduit la notion de système *partiellement* synchrone :

- Dans un premier modèle, il existe des bornes sur les délais de transmission des messages ou le temps nécessaire pour effectuer un pas de calcul. Cependant, ces bornes ne sont pas connues par les processus.
- Dans une autre version, le système se comporte de façon synchrone à partir d'un certain temps appelé *GST* (pour *Global Stabilization Time*).

Les auteurs étudient chacun des modèles associés aux paramètres de ces hypothèses (borne connue ou inconnue, stabilisation immédiate ou inéluctable) et montrent que certains problèmes à résoudre dans le modèle purement asynchrone le deviennent. D'autres travaux, en relation avec la recherche de la synchronie minimale nécessaire pour résoudre le consensus, étudient des restrictions spatiales du modèle partiellement synchrone [8, 9, 13, 83, 112]. Dans ces systèmes, seule une partie du système exhibe un comportement synchrone à partir d'un certain temps.

Dans le modèle *asynchrone temporisé* [40], l'hypothèse « une borne x apparaîtra à partir d'un certain temps » est remplacée par « une borne x apparaîtra infiniment souvent ». Ce modèle essaie de traduire l'observation que les systèmes réels oscillent entre périodes stables et périodes instables. Il généralise les modèles proposés dans [44, 45]. D'autres modèles raffinent les modèles ci-dessus. Par exemple, le modèle *quasi-synchrone* [10] impose des bornes connues sur la vitesse d'exécution, le délai de transfert des messages, le décalage temporel des horloges locales et la charge du système. Cependant, chacune de ces bornes peut être violée avec une probabilité p connue.

Ce bref survol montre qu'il existe une multitude de modèles qui correspondent à différents degrés d'incertitude temporelle. D'autre part, une modélisation fine d'un système réel nécessite un grand nombre de paramètres [10]. Dans cette thèse, nous choisissons le modèle purement asynchrone pour sa simplicité et sa généralité. En effet, tout résultat positif dans ce modèle s'étend aux modèles qui imposent une restriction sur l'asynchronie du système. Enfin, comme nous le verrons par la suite (paragraphe 1.2), la synchronie d'un modèle peut être abstraite au travers d'oracles appelés *détecteur de défaillances*. Au lieu de faire des hypothèses temporelles, nous enrichirons le modèle asynchrone avec des oracles (les détecteurs de défaillances) qui masquent des propriétés temporelles du système.

1.2 Détecteurs de défaillances

Comme nous l'avons vu, les processus sont sujets aux crashs ou pannes franches. La difficulté ou l'impossibilité de résoudre un problème donné provient essentiellement de la combinaison des défaillances et de l'absence de repères temporels. En effet, dans un modèle purement asynchrone, il est impossible de distinguer un processus défaillant d'un processus très lent. En particulier, il est impossible de résoudre le consensus dans un modèle purement asynchrone dans lequel un processus est susceptible de défaillir [48].

Pour contourner cette impossibilité, les chercheurs ont renforcé les hypothèses temporelles pour donner naissance aux modèles partiellement synchrones évoqués dans le paragraphe précédent. Quelle est l'influence de l'augmentation du degré de synchronie

du modèle sur la calculabilité ? Par exemple, considérons le consensus et l'un des modèles introduit par Dwork, Lynch et Stockmeyer [45]. Dans les exécutions de ce modèle, il existe des bornes sur les délais de transfert des messages ainsi que sur la durée maximale d'une étape de calcul. Bien que ces bornes ne soient pas connues a priori, il s'avère qu'il existe un algorithme de consensus qui tolère une minorité de processus défaillants ($t < \frac{n}{2}$).

Abstraire les hypothèses temporelles Ainsi, il est possible de résoudre le consensus si nous restreignons l'incertitude temporelle. De façon équivalente, le nombre de problèmes qu'il est possible de résoudre si le système est équipé d'un mécanisme qui donne des informations sur les défaillances augmente.

Supposons que les processus exécutent l'algorithme de détection de défaillances suivant [30, 45]. Périodiquement, chaque processus p_i émet un message « p_i est vivant » et attend ensuite des messages « de vie » des autres processus avant l'expiration d'un délai de garde. Si la minuterie expire avant la réception d'un message « p_j est vivant », p_i inclut p_j dans la liste des processus *suspects*. Si plus tard, p_i reçoit un message de p_j, il en déduit qu'il l'a soupçonné à tort et le retire donc de la liste des suspects. Il augmente également le délai de la minuterie pour éviter qu'une telle erreur ne se reproduise. Les propriétés de synchronie partielle du modèle impliquent l'existence d'un instant à partir duquel tous les processus défaillants seront soupçonnés en permanence tandis que les processus corrects ne seront plus jamais soupçonnés à tort.

L'information sur les défaillances produite par l'algorithme ci-dessus dans un système partiellement synchrone peut être abstraite de la façon suivante. Soit un modèle purement asynchrone dans lequel les processus ont accès à un *oracle* distribué. À chaque processus et à chaque instant, l'oracle fournit une liste de processus suspects. L'oracle garantit qu'il existe un instant à partir duquel (1) tous les processus défaillants seront toujours soupçonnés et (2) aucun processus correct ne sera jamais plus soupçonné. Cet oracle est le *détecteur de défaillances inéluctablement parfait* $\diamond\mathcal{P}$.

Quelle est la puissance de calcul dans le nouveau modèle enrichi ? Il a été montré qu'à l'aide de cet oracle en apparence très faible et peu fiable (les processus ne connaissent pas l'instant à partir duquel les listes de suspects sont pertinentes), il est possible de résoudre le consensus malgré les défaillances éventuelles d'une fraction des processus.

Capturer la synchronie minimale L'approche détecteurs de défaillances [30] consiste à enrichir le modèle asynchrone avec un oracle qui fournit des informations plus ou moins pertinentes, au lieu de restreindre explicitement l'asynchronie en faisant des hypothèses temporelles. À travers cette abstraction, on cherche à caractériser la « quantité de synchronie » minimale nécessaire et suffisante pour résoudre un problème donné.

Certains détecteurs sont comparables par l'intermédiaire de réductions algorithmiques. Un détecteur $\mathcal{D}$ est dit plus faible qu'un autre détecteur $\mathcal{D}'$ s'il existe un algorithme distribué fondé sur $\mathcal{D}'$ dont la sortie satisfait la spécification de $\mathcal{D}$. On notera $\mathcal{D}' \rightsquigarrow \mathcal{D}$ l'existence d'un tel algorithme. Intuitivement, les hypothèses temporelles abstraites par $\mathcal{D}$ sont plus faibles que celles capturées par $\mathcal{D}'$. Étant donné un problème

P impossible à résoudre dans un modèle purement asynchrone, cette notion d'ordre permet de définir (s'il existe) le plus faible détecteur $\mathcal{D}_{\min}$ pour ce problème. $\mathcal{D}_{\min}$ est minimal pour P s'il existe un algorithme qui résout P dans un modèle asynchrone muni d'un tel oracle et si pour tout détecteur $\mathcal{D}$ à l'aide duquel P peut être résolu, $\mathcal{D} \rightsquigarrow \mathcal{D}_{\min}$.

Étonnamment, pour certains problèmes un tel détecteur minimal existe et a été identifié. Le premier résultat de minimalité a été établi pour le consensus par Chandra, Hadzilacos et Toueg [29]. Plus tard, d'autres équipes ont identifié le détecteur minimal pour plusieurs problèmes fondamentaux en environnement réparti. Citons notamment l'implémentation d'un registre atomique dans le modèle à passage de messages, lorsque le nombre de processus défaillants n'est pas borné a priori ($t = n - 1$) [42], l'exclusion mutuelle tolérante aux défaillances [43], la validation atomique non bloquante [62, 79].

Les paragraphes suivants définissent formellement la notion d'oracle détecteurs de défaillances. Les définitions sont issues du travail fondateur de Chandra et Toueg [30].

1.2.1 Définition

Un détecteur de défaillances est un oracle qui fournit à chaque instant des informations plus ou moins pertinentes sur l'état des processus (vivant ou défaillant). Aucune contrainte n'est imposée sur l'encodage de cette information. Il est seulement requis que la spécification de la sortie de l'oracle dépende *uniquement des défaillances* (c'est-à-dire qu'un détecteur ne fournit pas d'indications sur les valeurs des variables locales des processus). D'un point de vue opérationnel, chaque processus p_i est muni d'une variable locale FD_i accessible seulement en lecture. Les valeurs successives de cette variable sont contrôlées par le détecteur de défaillances. Le processus p_i peut à tout instant interroger le détecteur en lisant la valeur courante de la variable FD_i.

Une classe de détecteur de défaillances est associée à un ensemble $\mathcal{R}$ (éventuellement infini), qui représente des valeurs potentiellement fournies par les détecteurs appartenant à cette classe. Un *historique de détection de défaillances* H est une fonction définie sur $\Pi \times \mathbb{T}$ à valeur dans $\mathcal{R}$. $H(i, \tau)$ représente l'information fournie par le détecteur au processus p_i à l'instant τ. Un détecteur de défaillances $\mathcal{D}$ à valeurs dans $\mathcal{R}_\mathcal{D}$ est une fonction qui associe à chaque motif de défaillances F un ensemble d'historiques de détection à valeurs dans $\mathcal{R}_\mathcal{D}$. Cet ensemble noté $\mathcal{D}(F)$ correspond à l'ensemble des historiques de détection autorisés par $\mathcal{D}$ pour le motif F. Lorsque p_i lit FD_i, il obtient par conséquent une valeur $d \in \mathcal{R}_\mathcal{D}$ qui encode de l'information relative au motif de défaillance courant. Remarquons qu'aucune hypothèse n'est faite sur l'ensemble des valeurs possibles $\mathcal{R}_\mathcal{D}$. Cependant, un détecteur ne peut fournir d'information sur le calcul effectué par un processus donné. À partir de FD_i, p_i peut au mieux déduire l'état (vivant ou défaillant) d'un processus.

À titre d'exemple, nous donnons la spécification des classes « mères » dont les détecteurs étudiés dans cette thèse sont issus.

- Le détecteur *parfait* $\mathcal{P}$ [30] fournit une liste de processus de « confiance » à chaque processus. Chaque processus est muni d'une variable TRUSTED_i qui contient un ensemble d'identités de processus supposés corrects. $\mathcal{P}$ garantit la *complétude forte* : au bout d'un certain temps les processus ne font plus confiance aux processus dé-

faillants et la *précision forte* : chaque processus fait toujours confiance à tous les processus corrects. Le détecteur est donc à valeur dans $\mathcal{R}_{\mathcal{P}} = 2^{\Pi}$ et étant donné un motif de défaillances F, les historiques de détection permis doivent vérifier :

$$\exists \tau \in \mathbb{T}, \forall i \in \Pi, \forall \tau' : (\mathit{Correct}(F) \subseteq H(i,\tau')) \wedge (\tau' \geq \tau \Rightarrow H(i,\tau') = \mathit{Correct}(F))$$

C'est-à-dire de façon plus opérationnelle :

$$\exists \tau \in \mathbb{T}, \forall i \in \Pi, \forall \tau' : (\mathit{Correct} \subseteq \textsc{trusted}_i^{\tau'}) \wedge (\tau' \geq \tau \Rightarrow \textsc{trusted}_i^{\tau'} = \mathit{Correct})$$

– De même que $\mathcal{P}$, le détecteur *inéluctablement faible* $\diamond\mathcal{S}$ [30] fournit une liste d'identités de processus de confiance supposés corrects. Cependant les garanties offertes par ce détecteur sont plus faibles. Au bout d'un certain temps, un détecteur $\diamond\mathcal{S}$ fournit une liste qui ne contient pas d'identité de processus fautifs (complétude forte) et il existe un processus correct à qui tous les processus font confiance (*précision faible inéluctable*). Formellement, étant donné un motif de défaillances F, $H \in \diamond\mathcal{S}(F) \Leftrightarrow$

$$\exists \tau \in \mathbb{T}, \exists c \in \mathit{Correct}(F), \forall i \in \Pi, \forall \tau' \geq \tau : H(i,\tau') \subseteq \mathit{Correct}(F) \wedge c \in H(i,\tau')$$

ou, en supposant que le détecteur contrôle le contenu des variables $\textsc{trusted}_i$:

$$\exists \tau \in \mathbb{T} : \Big(\bigcup_{\tau' \geq \tau, i \in \Pi} \textsc{trusted}_i^{\tau'} \subseteq \mathit{Correct}(F) \Big) \wedge \Big(\bigcap_{i \in \Pi, \tau' \geq \tau} \textsc{trusted}_i^{\tau'} \neq \emptyset \Big)$$

– Le détecteur *leader* Ω [29] fournit à chaque processus une identité. Il existe un instant à partir duquel la même identité est fournie à tous les processus et cette identité correspond à un processus correct. Soit F un motif défaillance

$$H \in \Omega(F) \Leftrightarrow \exists \tau \in \mathbb{T}, \exists c \in \mathit{Correct}(F), \forall i \in \Pi, \forall \tau' \geq \tau : c = H(i,\tau')$$

On notera $\textsc{leader}_i$ la variable contrôlée par un détecteur Ω sur le processus p_i.

Les spécifications de ces trois classes font apparaître certaines relations entre ces trois détecteurs. Par exemple, étant donné un motif F, tout historique $H \in \Omega(F)$ appartient à $\diamond\mathcal{S}(F)$. De même, l'ensemble des historiques qui satisfont la spécification du détecteur $\mathcal{P}$ est inclus dans $\diamond\mathcal{S}(F)$. Intuitivement, le détecteur $\mathcal{P}$ est « plus fort » que les détecteurs $\diamond\mathcal{S}$ et Ω.

Réduction Nous pouvons maintenant formaliser la notion de réduction entre détecteurs de défaillances déjà esquissée précédemment. Soient $\mathcal{D}$ et $\mathcal{D}'$ deux détecteurs de défaillances. S'il existe un algorithme $T_{\mathcal{D}' \to \mathcal{D}}$ qui transforme $\mathcal{D}$' en $\mathcal{D}$, $\mathcal{D}$ est dit *plus faible* que $\mathcal{D}'$ (noté $\mathcal{D}' \rightsquigarrow \mathcal{D}$). Un tel algorithme maintient sur chaque processus p_i une variable $\textsc{out\_d}_i$ dont les valeurs successives sont compatibles avec le détecteur $\mathcal{D}$. Par compatible, nous entendons que pour tout motif de défaillances F et pour toute exécution dans laquelle les défaillances suivent F, l'historique de détection $H'(i,\tau)$, c'est-à-dire les valeurs successives de $\textsc{out\_d}_i$ appartient à $\mathcal{D}'(F)$. Notons que $T_{\mathcal{D}' \to \mathcal{D}}$ n'émule pas nécessairement l'ensemble des historiques de détection de défaillances de $\mathcal{D}$. L'algorithme doit seulement assurer que tous les historiques de détection qu'il produit appartiennent à $\mathcal{D}'$.

1.2.2 Modèles choisis et notations

Dans cette thèse, nous nous plaçons dans un modèle dans lequel n processus (n est connu *a priori*) s'exécutent de manière *totalement asynchrone*. Ces processus communiquent par *passage de messages* ou par l'intermédiaire d'une *mémoire partagée*. Dans ce dernier cas, la mémoire est composée de *registres atomiques* à *écrivain unique*. Le médium de communication, c'est-à-dire la transmission des messages ou la mémoire partagée est fiable. Par contre, les processus sont susceptibles de connaître des défaillances. Le seul type de défaillance que nous considérons est la panne franche, c'est-à-dire l'arrêt inopiné d'un processus. Au cours d'une exécution, le nombre maximal de processus qui tombent en panne est noté t. Nous enrichirons le modèle avec des oracles de type *détecteurs de défaillances*.

Soit $\mathcal{C}$ un détecteur de défaillances. Nous utiliserons les notations suivantes :

- $\mathcal{MP}_{n,t}[\mathcal{C}]$ désigne le modèle asynchrone avec communication par passage de messages dans lequel les processus ont accès à un détecteur $\mathcal{C}$. Les indices n et t désignent respectivement le nombre de processus et le nombre maximal de pannes pouvant survenir dans chaque exécution.
- $\mathcal{SM}_{n,t}[\mathcal{C}]$ désigne le modèle asynchrone avec communication par registres muni d'un détecteur $\mathcal{C}$. Les indices n et t ont la même signification que précédemment.

Par extension $\mathcal{MP}_{n,t}[\emptyset]$ et $\mathcal{SM}_{n,t}[\emptyset]$ désignent des modèles asynchrones pures, c'est-à-dire qui ne bénéficient d'aucun oracle.

1.3 Outillage algorithmique

Nous nous efforçons d'adopter une approche algorithmique. Comme noté dans [22],

> *Identifying high-level constructs is essential for advancing the art of distributed algorithm design.*

Cette partie présente des constructions de haut niveau dans le modèle à mémoire partagée. Ces constructions ont pour but de fournir aux différents processus des vues d'un état global courant « qui ne diffèrent pas trop ». Il est important de souligner que toutes les constructions présentées ci-après n'augmentent pas la puissance calculatoire du modèle de base. Pour chacune d'entre elles, il existe un algorithme qui tolère un nombre arbitraire de défaillances ($t = n - 1$) dans le modèle $\mathcal{SM}_{n,t}[\emptyset]$.

Synchronisation de vues Nous avons progressivement enrichi le modèle à passage de messages $\mathcal{MP}_{n,t}[\emptyset]$, d'abord avec la primitive broadcast()/deliver() puis avec la primitive R_bcast()/R_deliver(). Ces abstractions qui offrent une sémantique de plus en plus riche facilitent la conception d'algorithmes distribués. De plus chacune d'entre elles peut être construite à partir de la primitive de base du modèle send()/receive(). Ces abstractions concernent la diffusion globale d'informations à travers tout le système. Un appel à R_bcast() (ou broadcast()) introduit un nouveau bloc d'informations dans le système tandis qu'en appelant R_deliver() (ou deliver()) un processus enrichit sa connaissance sur l'état global.

Ce type de problème (diffusion de l'état local/connaissance de l'état global) a largement été étudié dans le cadre du modèle mémoire partagée $\mathcal{SM}_{n,t}[\emptyset]$. Ci-après, nous dressons un bref panorama des primitives connues qui permettent de calculer une vue de l'état du système avec différents critères de cohérence.

Ces primitives sont toujours définies par un couple de fonction (écrire/lire) ; la première permet de stocker une information à destination des autres processus (diffusion de l'état local), la deuxième a pour but d'obtenir les valeurs stockées par les autres processus. Pour chacune des primitives définies ci-après, il existe une implémentation qui tolère un nombre arbitraire de défaillances ($t = n - 1$) à partir de registres atomiques à écrivain unique. Puisque notre étude s'intéresse principalement à la calculabilité, nous usons sans réserve de ces abstractions dans les chapitres suivants.

Pour simplifier les définitions, nous nous plaçons dans le contexte suivant : chaque processus p_i possède une valeur v_i appartenant à un alphabet $\mathcal{V}$ et désire connaître la valeur des autres processus. Pour une exécution donnée, les valeurs initiales sont représentées par un vecteur $I \in \mathcal{V}^n$ tel que $I[i] = v$ si p_i possède initialement la valeur v ou $\perp$ si p_i ne participe pas. Une vue du système obtenue par un processus p_i est un ensemble sm_i de paires $< j, v_j >$ où v_j est la valeur initiale de p_j. Par un léger abus de notation, nous noterons pour $j \in sm_i$ ou $v_j \in sm_i$ lorsque $< j, v_j > \in sm_i$.

1.3.1 Collecte

Une primitive de *collecte simple*[2] [68, 121] garantit que chaque processus obtient des valeurs au moins aussi récentes que celles de sa dernière lecture. Formellement, la sémantique de la paire write(v)/collect() est la suivante (voir par exemple [6]).

Définition 1.2 (Collecte) *Si un processus p_i exécute l'opération op : $sm_i \leftarrow$ collect() alors :*

1. $\forall j : j \notin sm_i \Rightarrow$ *aucune opération* write(v) *de p_j ne précède op ;*
2. $\forall j :< j, v >\in sm_i \Rightarrow$ *v est la valeur écrite par p_j lors d'une opération wop :* write(v)*, telle que wop précède op et qu'aucune autre opération* write(v') *par p_j n'est intercalée entre wop et op ;*
3. *Si p_i exécute une seconde opération op' : $sm'_i \leftarrow$ collect() après op alors $sm_i \subseteq sm'_i$.*

De même que la primitive broadcast()/deliver(), l'implémentation de write()/collect() est très simple. Il suffit de lire successivement le contenu des registres qui composent la mémoire (Figure 1.2). Pour annoncer sa valeur, p_i écrit simplement cette valeur dans le registre $M[i]$. L'atomicité des opérations de bas niveau sur les registres garantit les trois propriétés de la spécification.

La primitive précédente n'offre que peu de garanties quant à l'ordre respectif des lectures lorsqu'elles sont concurrentes. Dans l'exemple décrit dans la figure 1.3, p_1 écrit en premier. L'exécution de l'opération collect() par p_1 est concurrente avec les écritures de p_2 et p_3. p_2 ne voit pas l'écriture de p_3 et retourne $sm_2 = \{1, 2\}$. L'écriture de

[2] *collect* dans la littérature.

```
operation collect()
  sm_i ← ∅
  foreach j ∈ Π do v ← M[j].read()
    if v ≠ ⊥ then sm_i ← sm_i ∪ {< j, v >} endif
  enddo
  return(sm_i)
```

FIG. 1.2 – La primitive collect()

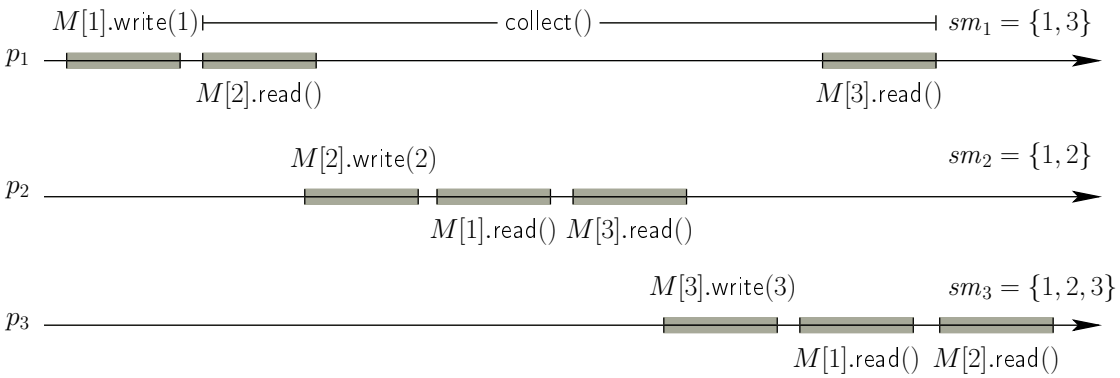

FIG. 1.3 – Exécution non linéarisable, $n = 3$

p_3 est physiquement la dernière, il retourne nécessairement $\{1, 2, 3\}$. Enfin p_1 renvoie $sm_1 = \{1, 3\}$. Les deux ensembles $\{1, 3\}$ et $\{1, 2\}$ sont incomparables, ce qui implique qu'il n'est pas possible de placer écritures et lectures (collect()) sur un axe de temps commun. Or, « placer les opérations sur un axe de temps commun » en respectant leur sémantique simplifierait le raisonnement sur les exécutions réparties. En effet, au lieu d'intervalles de temps qui s'intersectent éventuellement, on aurait une vision séquentielle des opérations successives, chaque opération donnant l'illusion de s'exécuter instantanément. Ce concept est connu sous le nom de *linéarisation*.

Linéarisation Ce concept a été introduit par Herlihy et Wing [76] afin de simplifier le raisonnement lors de la conception d'algorithmes distribués. Informellement, un objet est *linéarisable* si sa spécification peut s'exprimer par une machine séquentielle (généralement hors du système).

Dans le cas d'une paire de primitives (annonce d'une valeur/lecture de l'ensemble des valeurs), la linéarisation stipule qu'il existe un axe de temps commun et une date pour chacune des opérations d'écriture et de lecture [76] :

> *Linearizability provides the illusion that each operation takes effect instantaneously at some point between its invocation and its response. (...) It facilitates certain kinds of formal (and informal) reasoning (...) It states that certain interleavings cannot occur.*

Un registre atomique est clairement linéarisable car il n'est modifiable que par un

seul processus séquentiel et les opérations qu'il supporte sont linéarisables. Par contre, comme le montre la figure 1.3, un vecteur de registres muni des opérations write() et collect() n'est pas linéarisable.

1.3.2 Collecte ordonnée ou Atomic snapshot

Nous souhaitons renforcer la primitive (lire/écrire) pour obtenir un objet linéarisable. La *collecte ordonnée* ou *snapshot* requiert que les ensembles renvoyés soient ordonnés par inclusion, en plus des contraintes de (write()/collect()). Ce problème a été posé et résolu par plusieurs équipes de recherche : Afek *et al.* [1] définissent *atomic snapshot*, Aspnes et Herlihy [70] parlent d'*atomic scan* et Anderson [11] le nomme *composite register*. Ces travaux donnent des spécifications différentes ainsi que des implémentations dans un modèle à registres. La collecte ordonnée ou atomic snapshot est définie par la paire (write()/snap()) qui vérifie la spécification suivante :

Définition 1.3 (Collecte ordonnée ou Atomic snasphot) *La paire write()/snap() de la collecte ordonnée satisfait : Supposons que chaque processus effectue successivement write(v_i) ; $sm_i \leftarrow$ snap(). Alors les ensembles sm_i vérifient :*

1. Auto-inclusion $\forall p_i : i \in sm_i$;

   (sm_i contient la valeur écrite par p_i)

2. Comparaison $\forall p_i, p_j : (sm_i \subseteq sm_j) \vee (sm_j \subseteq sm_i)$.

   (Les ensembles sm_i sont ordonnés par la relation d'inclusion)

L'ordre par la relation d'inclusion sur les vues retournées supprime les exécutions qui comportent des inversions entre des valeurs anciennes et des valeurs récentes (comme dans l'exemple de la figure 1.3). De plus, l'exécution est linéarisable en se basant sur l'ordre des ensembles retournés. Par exemple, une linéarisation possible peut être obtenue à l'aide de l'algorithme suivant :

- ranger les ensembles dans l'ordre $sm_1 \subseteq sm_2 \subseteq \cdots \subseteq sm_k$;
- les écritures correspondant aux valeurs de sm_1 sont placées au temps $\tau = 1$, les écritures des valeurs dans $sm_2 - sm_1$ sont placées au temps $\tau = 2$, etc.

Il existe plusieurs implémentations sans attente[3] d'un objet atomic snapshot AS à partir de registres atomique à écrivain unique. Le meilleur algorithme à ce jour dû à Attiya et Rachman [18] possède une complexité en $O(n \log(n))$ en nombre d'appels aux opérations write(v)/read() de base.

La figure 1.4 représente les entrelacements possibles pour 3 processus. Chaque processus p_i écrit son index et calcule une vue des index écrits dans la mémoire partagée R :

$$R[i].\mathsf{write}(i); view_i \leftarrow R.\mathsf{snap}()$$

[3]Un algorithme qui tolère un nombre arbitraire de défaillances ($t = n - 1$) ne peut comporter d'instructions d'attente d'informations en provenance d'autres processus. La notion de calcul sans attente sera discutée plus en détails dans le paragraphe 1.4.3

L'état final du processus p_i peut être représenté par le couple $(p_i, view_i)$. Chaque sommet du graphe de la figure 1.4 représente un état final possible. De plus, une arête est tracée entre les sommets $s = (p, view)$ et $s' = (p', view')$ si les états correspondants sont compatibles, c'est-à-dire qu'il existe une exécution dans laquelle le processus p calcule la vue *view* et le processus p', *view*$'$.

Puisque les processus sont susceptibles d'être défaillants, il est possible que dans certaines exécutions seuls certains processus obtiennent une vue (les autres tombant en panne avant de renvoyer un snapshot). Les exécutions où seul un processus participe (les autres processus tombent en panne avant d'exécuter la moindre modification de la mémoire partagée) sont représentées par l'un des trois coins du triangle extérieur. Ainsi, le sommet étiqueté (p_1,{1}) représente l'exécution dans laquelle seul p_1 participe. De même, les exécutions à deux participants sont représentées par un segment (deux sommets reliés par une arête) sur l'un des bords du triangle. Enfin, les triangles intérieurs représentent les exécutions à trois participants. La figure 1.5 donne des linéarisations possibles des exécutions qui correspondent aux deux triangles grisés dans la figure 1.4.

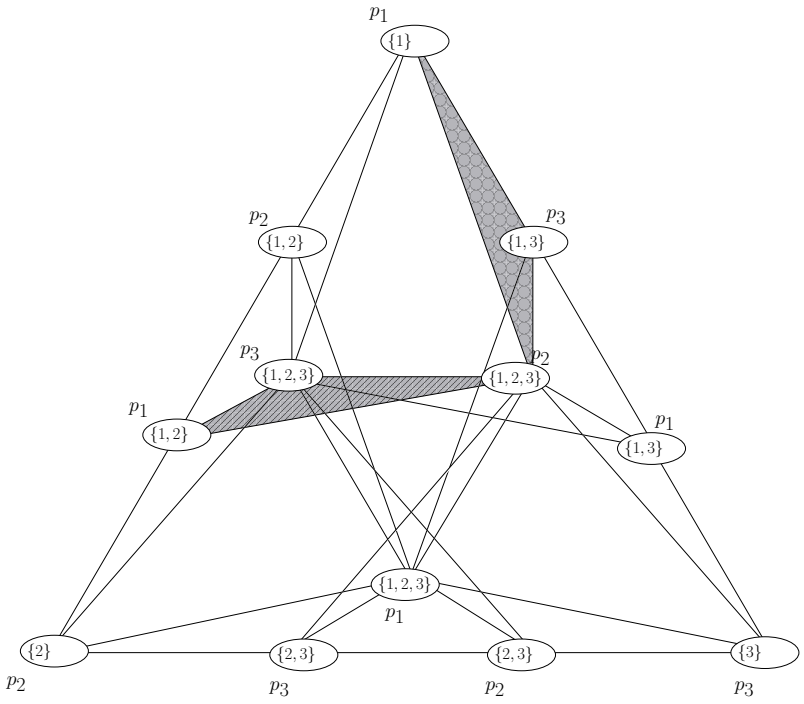

FIG. 1.4 – Entrelacements des opérations write()/snap(), $n = 3$

Est-il possible de contraindre d'avantage la primitive (lire/écrire) ? Plus la primitive restreint le nombre d'entrelacements possibles, plus le raisonnement et la construction d'algorithmes sont facilités. Le paragraphe suivant présente la *collecte ordonnée immédiate* introduit par Borowsky et Gafni.

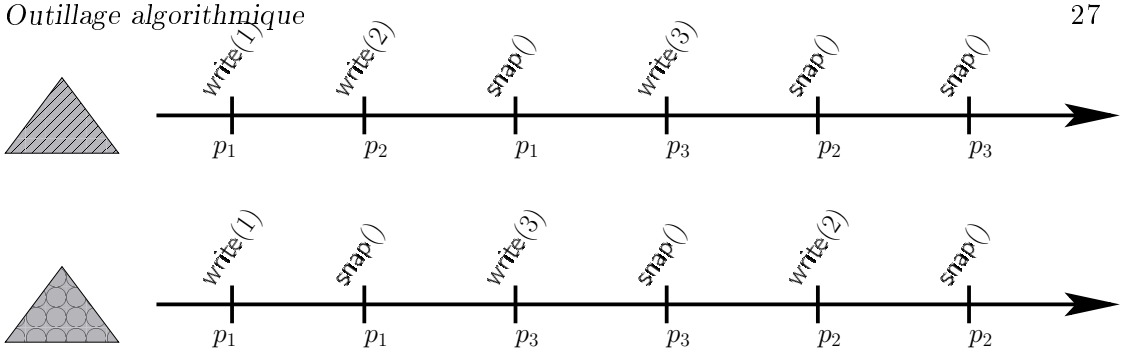

FIG. 1.5 – Linéarisation des opérations write()/snap() de la figure 1.4

1.3.3 Collecte ordonnée immédiate (Immediate Snapshot)

Dans [21, 22], Borowsky et Gafni introduisent le problème de la collecte ordonnée immédiate. Au lieu de séparer la lecture et l'écriture, ces deux opérations sont fusionnées en une seule notée write_snap(). Un appel write_snap(v) par p_i renvoie un snapshot atomique sm_i. Ce snapshot est *immédiat* dans le sens où la primitive donne l'illusion que p_i exécute write(v) immédiatement suivi de $sm_i \leftarrow$ snap().

Définition 1.4 (Collecte ordonnée immédiate ou Immediate Snapshot) *La primitive write_snap() de la collecte ordonnée immédiate satisfait : si l'on suppose que chaque processus exécute $sm_i \leftarrow$ write_snap(v_i). Alors les ensembles sm_i vérifient :*

1. Auto-inclusion $\forall p_i : i \in sm_i$;

   (sm_i contient la valeur écrite par p_i)

2. Comparaison $\forall p_i, p_j : (sm_i \subseteq sm_j) \vee (sm_j \subseteq sm_i)$;

   (Les ensembles sm_i sont ordonnées par la relation d'inclusion)

3. Immédiateté $\forall p_i, p_j : j \in sm_i \Rightarrow sm_j \subseteq sm_i$.

   (Si sm_i contient la valeur de p_j alors sm_i contient le snapshot retourné par p_j)

La contrainte d'immédiateté restreint l'espace des exécutions possibles, comme le montre la figure 1.6. Dans ce dessin sont représentés les états possibles des processus après un appel à la primitive write_snap(i), avec les mêmes conventions employées dans la figure 1.4. Par exemple, l'exécution qui correspond au triangle hachuré dans la figure 1.4 ne peut se produire. En contrepartie, les exécutions ne sont plus linéarisables, mais *set-linéarisables* [105] : en un seul point, plusieurs opérations de processus différents apparaissent logiquement comme s'exécutant simultanément en un seul point. Par exemple, dans l'exécution correspondant au triangle central de la figure 1.6, les opérations write_snap() des trois processus sont logiquement simultanées.

L'échelle de Borowsky-Gafni [22] Un algorithme qui implémente sans attente (c'est-à-dire qui tolère un nombre arbitraire de pannes, $t = n - 1$; la notion de calcul sans attente sera précisée dans le paragraphe 1.4.3) un objet snapshot immédiat

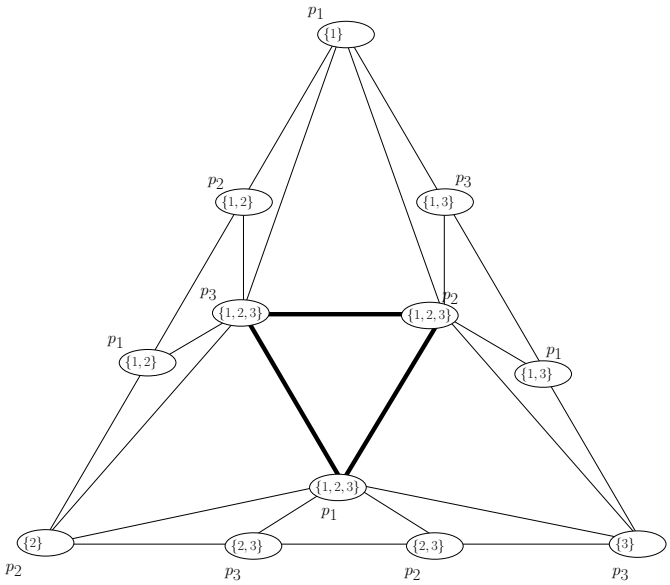

FIG. 1.6 – write_snap() : toutes les vues possibles ($n = 3$)

init : $LEVEL[1,\ldots,n] \leftarrow [n+1,\ldots,n+1]$; $V[1,\ldots,n] \leftarrow [\perp,\ldots,\perp]$

operation write_snap(v_i)
  $V[i] \leftarrow v_i$;
  repeat $LEVEL[i] \leftarrow LEVEL[i] - 1$;
    $level_i \leftarrow LEVEL$.collect() ;
    $view_i \leftarrow \{j : (< j, \ell_j > \in level_i) \wedge (\ell_j \leq LEVEL[i])\}$
  until $(|view_i| \geq LEVEL[i])$ **endrepeat** ;
  $sm_i \leftarrow \{< j, V[j] >: j \in view_i\}$; return(sm_i)

FIG. 1.7 – Collecte ordonnée immédiate [22]

dans le modèle à mémoire partagée est décrit dans la figure 1.7. L'algorithme utilise un tableau partagé $LEVEL[1,\ldots,n]$ constitué de n registres atomiques à écrivain unique.

L'intuition derrière l'algorithme est la suivante. Les processus descendent un escalier de $n+1$ degrés, marches après marches. L'algorithme vise à répartir les processus sur les marches de telle sorte que tous les processus qui s'arrêtent au même *niveau* retournent la même vue. Les processus sont initialement positionnés au sommet de l'escalier (niveau $n+1$). Le tableau $LEVEL$ enregistre la position des différents participants. Lorsque p_i atteint le niveau ℓ, il l'indique en positionnant $LEVEL[i]$ à ℓ.

Lorsqu'il est au niveau ℓ, le processus p_i n'a le droit de descendre qu'à la condition qu'il y ait de la place dans les niveaux inférieurs. Plus précisément, p_i dispose d'espace pour descendre si les processus qui occupent les marches $\leq \ell$ sont en nombre $< \ell$. Pour vérifier s'il a atteint sa position finale, p_i collecte la position courante des autres

processus. Les processus situés sur la même marche ou sur les marches inférieures sont les processus p_j tels que $LEVEL[j] \leq \ell$. Les identités de ces processus sont stockées dans l'ensemble $view_i$. Si $|view_i| < \ell$, p_i continue sa descente. Dans le cas contraire, p_i s'arrête au niveau ℓ. Le snasphot immédiat qu'il retourne contient les valeurs de tous les processus qu'il voit, i.e., $\{v_j : j \in view_i\}$.

1.4 Coordination dans les systèmes distribués

Le problème de coordonner des processus concurrents demeure l'un des problèmes fondamentaux du calcul distribué tolérant aux fautes. Dans ce contexte, le problème du consensus a été intensivement étudié. C'est une brique fondamentale pour la résolution de nombreux problèmes distribués.

Dans ce problème, chaque processus est initialement muni d'une valeur privée issue d'un ensemble quelconque et doit au bout d'un certain temps décider une valeur. La spécification décrit quelles sont les décisions légales en fonction des valeurs initiales et requiert également que les processus corrects décident :

Définition 1.5 (Spécification du consensus)
Validité *La décision de chaque processus est la valeur initiale d'un processus ;*
Accord *Les décisions de tous les processus sont identiques ;*
Terminaison *Les processus corrects décident.*

Si les valeurs initiales sont booléennes, ce problème est appelé *consensus binaire*. Fisher, Lynch et Paterson ont montré [48] que ce problème n'a pas de solution déterministe dans un environnement asynchrone dans lequel au moins un processus est susceptible de tomber en panne.

1.4.1 Problèmes étudiés

Dans cette thèse, nous étudions les relations entre différents problèmes de coordination faiblement contrainte. Du point de vue de la calculabilité en environnement asynchrone avec défaillances, ces problèmes sont strictement plus faciles que le consensus (*subconsensus tasks*). Autrement dit, étant donné une solution à l'un de ces problèmes, il n'est pas possible de résoudre le consensus à l'aide de celle-ci. À l'inverse, par l'universalité du consensus [69], une solution au problème du consensus implique l'existence d'une solution pour chacun des problèmes considérés. Toutefois, ces problèmes sont non triviaux : il n'existe pas en général d'algorithmes déterministes pour les résoudre dans les modèles asynchrones avec défaillances par panne franche des processus.

Nous présentons brièvement les problèmes étudiés dans ce document. Les spécifications détaillées seront données dans le chapitre 2

Consensus ensembliste Le *consensus ensembliste* ou (n, k)-accord généralise le consensus en relâchant la contrainte d'accord. Au lieu d'obliger les processus à décider la même valeur, le (n, k)-accord requiert que l'ensemble des valeurs décidées par les n processus composant le système soit de cardinal au plus k :

Définition 1.6 (Accord ensembliste)
k-Accord *Au plus k valeurs sont décidées.*

Ce problème a été introduit par Chaudhuri [31, 32] dans le but d'étudier comment le « degré de coordination » (capturé par le paramètre k) des problèmes influe sur leur calculabilité. Elle montre qu'il existe une solution lorsque $t < k$ (où t dénote le nombre maximal de pannes) et conjoncture qu'il est impossible de résoudre ce problème lorsque $t \geq k$. Ce résultat sera établi plus tard par trois équipes indépendantes [75, 21, 115]. Un aperçu des techniques employées est présenté dans le paragraphe 1.4.5.

Décision de comité Ce problème dérive également du consensus. Alors que l'accord ensembliste généralise le consensus en affaiblissant la propriété de « sûreté », la décision en comité [57, 58] rend plus facile le consensus en relaxant la propriété de « vivacité ». Les processus sont impliqués dans la résolution simultanée de plusieurs consensus, appelés comité. La terminaison est relaxée dans le sens suivant : chaque processus doit décider dans au moins l'un des consensus. Il n'est ni requis que tous les processus choisissent la même instance ni que dans chaque instance, il existe un processus qui décide. Nous noterons (n, k)-comité ce problème lorsqu'il est défini pour n processus cherchant à décider en parallèle dans k comités. Par rapport au consensus, le « degré de liberté » supplémentaire découle du choix laissé libre de l'instance du consensus dans lequel chaque processus décide. Différemment, le degré de liberté offert par le consensus ensembliste réside en l'autorisation de choisir collectivement k valeurs.

Test&set ensembliste La primitive de synchronisation classique *test&set* permet de lire et modifier une variable en une seule opération atomique. Dans le cas d'une variable binaire initialisée à 1, une opération test&set renvoie la valeur précédente du bit et le met à 0. La puissance de calcul de cette primitive est capturée par le problème suivant, qui n'a pas d'entrées : il s'agit de décider une valeur $\in \{0, 1\}$ telle que, dans toute exécution, il existe un processus et un seul qui décide 1 (ce processus est le *gagnant* dans l'exécution considérée, les autres processus sont dits perdants). L'absence d'entrées rend ce problème strictement plus faible que le consensus binaire. Intuitivement, il n'est pas possible d'utiliser une solution au test&set pour communiquer de l'information entre les processus. Au contraire, dans le consensus binaire, les processus exercent un certain contrôle sur la sortie grâce au choix libre des propositions et à la validité de la valeur décidée. D'ailleurs, ce contrôle est suffisant pour construire une solution au consensus multi-valué [104, 119].

Cependant, il existe un algorithme simple qui résout le consensus fondé sur une solution au test&set dans un système asynchrone composé de $n = 2$ processus : décider la valeur proposé par le gagnant. Cette solution n'est correcte que si $n = 2$. En effet, si $n > 2$, un processus perdant n'est pas capable d'identifier le processus gagnant (par exemple, celui-ci tombe en panne avant d'annoncer qu'il a gagné).

Malgré le caractère peu puissant du test&set, il est possible de construire une famille de problème strictement plus faibles en relâchant la contrainte sur le nombre de processus gagnants. Dans le *test&set ensembliste* [21], le nombre de processus gagnants

dans chaque exécution est au moins 1 et au plus k. Cette famille abstrait des schémas de coordination qui ont une sémantique très faible. Ce problème, défini pour n processus, sera noté (n, k)-test&set.

Renommage Dans ce problème, les processus possèdent initialement un nom unique dans un vaste espace de nommage. Ils doivent décider un nouveau nom dans un espace plus petit. Pour éviter les solutions triviales, la solution doit être indépendante de l'index des processus. Une application possible de ce problème est l'augmentation de l'efficacité d'algorithmes dont la complexité dépend de l'espace de nommage des processus. En effet, l'exécution préalable d'une phase de renommage réduira la taille de l'espace de nommage en entrée de l'algorithme dont on souhaite améliorer la complexité. La spécification d'un problème de renommage impose une contrainte sur la taille l'espace final des noms. Le renommage est dit *adaptatif* si cette taille dépend du nombre de processus participant. Le problème du renommage, défini pour n processus, dans lequel p processus ($1 \leq p \leq n$) doivent acquérir un nouveau dans l'intervalle $[1, \ldots, h(p)]$, où h est une fonction de $\mathbb{N}$ à valeurs dans $\mathbb{N}$ sera noté (n, h)-renaming.

1.4.2 Forme générale d'un problème de décision

Les problèmes définis ci-dessus sont des représentants d'une classe importante de problèmes : les *problèmes de décision*. Chacun des n processus possède initialement une valeur connue de lui seul. Ces processus effectuent ensuite un certain nombre d'étapes de communication et de calculs locaux pour choisir une valeur finale qui dépend des contraintes du problème ainsi que de l'ensemble des valeurs initiales.

Les problèmes de décision sont supposés modéliser des systèmes réactifs comme des bases de données, systèmes de fichiers ou encore systèmes de contrôle aérien. Une valeur initiale représente de l'information en provenance de l'extérieur. Par exemple, il s'agit d'une valeur fournie par un capteur ou d'une chaîne de caractères entrée au clavier par l'utilisateur. Une valeur finale modélise une action qui affecte l'extérieur telle la décision irrévocable de valider une transaction.

Soit un système comportant n processus dont au plus t ($0 < t < n$) d'entre eux sont potentiellement défaillants. Dans un problème de décision, tous les processus participant au protocole proposent une valeur appartenant à un alphabet $\mathcal{V}$.

Définition formelle L'état global initial du système peut être représenté comme un vecteur $\mathcal{V}_{in}$ de dimension n sur l'alphabet $\mathcal{V} \cup \perp$.

$$V_{in}[i] = \begin{cases} \perp & \text{si } p_i \text{ ne participe pas} \\ v_i & \text{si } p_i \text{ propose la valeur } v_i \in \mathcal{V} \end{cases}$$

En utilisant cette représentation, une exécution comportant f ($0 \leq f \leq t$) processus défaillants définit un vecteur initial V_{in} qui contient *au plus* f fois la valeur par défaut $\perp$.

De même, l'état global en sortie est représenté par un vecteur V_{out} sur l'alphabet $\mathcal{V}_{out} \cup \perp$ éventuellement différent de l'alphabet d'entrée.

$$V_{out}[i] = \begin{cases} \perp & \text{si } p_i \text{ est défaillant} \\ v_i & \text{si } p_i \text{ a renvoyé la valeur } v_i \in \mathcal{V}_{out} \end{cases}$$

Définition 1.7 (problème de décision) *Un problème de décision est un triplet* $(\Delta, \mathcal{I}, \mathcal{O})$ *tel que*

- $\mathcal{I}$ *est l'ensemble des états globaux autorisés en entrée*
- $\mathcal{O}$ *est l'ensemble des états globaux autorisés en sortie*
- $\Delta \subseteq \mathcal{I} \times \mathcal{O}$ *est la spécification du problème, telle que* $(I, J) \in \Delta \Rightarrow (I[i] = \perp \Rightarrow J[i] = \perp)$

La contrainte sur la relation Δ indique simplement qu'un processus qui ne participe pas au protocole ne peut pas décider. Les problèmes considérés dans le chapitre 2 sont des problèmes de décision. À titre d'exemple, la spécification du consensus binaire à deux processus est donnée dans la figure 1.8 pour $t = 1$.

I	O
(1,1)	(1,1)
(1,⊥)	(1,⊥)
(⊥,1)	(⊥,1)
(0,⊥)	(0,⊥)
(⊥,0)	(⊥,0)
(0,0)	(0,0)
(1,0)	(1,1), (0,0)
(0,1)	(1,1), (0,0)

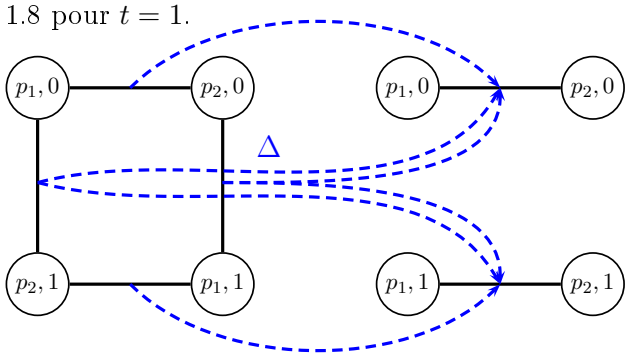

FIG. 1.8 – Spécification du consensus binaire pour deux processus

1.4.3 Calcul tolérant les défaillances

Pour résoudre un problème de décision P, les processus exécutent un *algorithme distribué* ou *protocole*. Un tel algorithme décrit quelles sont les informations à envoyer et les calculs locaux à exécuter. Chaque processus p_i est muni d'une variable dec_i locale qui ne supporte qu'une seule écriture. Cette variable, initialisée à $\perp$, est destinée à recevoir la décision finale de p_i.

Un protocole $\mathcal{A}$ est t-résilient s'il résout le problème P en dépit de t défaillances. Plus précisément, pour une exécution donnée, notons I le vecteur des valeurs initiales et J^τ le vecteur des valeurs des variables dec_i à l'instant τ. Soit $(\Delta, \mathcal{I}, \mathcal{O})$ la spécification de P. $\mathcal{A}$ est une solution t-résiliente pour le problème P si dans toute exécution telle que :

- au plus $f \leq t$ processus tombent en panne et
- $I \in \mathcal{I} \wedge |\{i : I[i] = \perp\}| \leq f$,

il existe un instant τ_d qui dépend de l'exécution tel que

- $\forall \tau \geq \tau_d : \Delta(I, J^\tau)$.

Calcul sans attente Dans le cas particulier $t = n - 1$, un algorithme t-résilient est dit *sans attente* (*wait-free*). Intuitivement, un tel environnement proscrit l'utilisation d'instruction d'attente d'informations en provenance d'autre processus. En effet, comme chaque processus est susceptible de tomber en panne, l'attente d'un message ou d'une écriture en provenance d'un autre processus risque de ne jamais terminer.

De façon équivalente, il existe une solution sans attente au problème P s'il existe un algorithme qui assure que tous les processus corrects décident en respectant la spécification de P en un nombre fini d'étapes de calcul. De plus, si le système est constitué d'un nombre fini de processus, il existe une borne globale (c'est-à-dire qui ne dépend pas de l'exécution) sur le nombre d'étapes de calcul.

Définition 1.8 (Calcul sans attente) *Un algorithme est dit sans attente s'il existe une borne supérieure (pour toutes les exécutions) du nombre d'opérations effectuées par un processus donné.*

En effet, puisque l'algorithme tolère au plus $n - 1$ processus défaillants parmi n, il n'est pas possible que le code contienne des boucles d'attente dont la terminaison dépend d'une propriété faisant intervenir d'autres processus. Le code est donc déroulable et de taille finie puisque chaque processus correct décide dans toute exécution en un temps fini.

1.4.4 Réduction

Nous nous intéressons aux relations entre certains problèmes de décision. En particulier, nous étudions des questions de la forme suivante : Étant donnée une solution « boîte noire » au problème P, est-il possible de résoudre un autre problème P'. Un moyen simple d'étudier cette question est d'enrichir le modèle avec des objets partagés qui offre une solution au problème P.

Notion d'objets Un objet est une structure de données partagée d'un certain *type*. Le type d'un objet décrit l'ensemble des états possibles des objets de ce type, l'ensemble des opérations qu'ils supportent ainsi que les réponses que ces objets renvoient lors d'interaction avec les processus. À chaque instant, un objet O est dans un certain état et lorsque qu'un processus exécute une opération sur O, son état varie et il retourne une réponse. Par exemple, un objet S de type snapshot (cf. definition 1.3) supporte deux opérations snap() et write(). Un tel objet mémorise un vecteur de valeurs. Les processus peuvent examiner le tableau pour apprendre les valeur contenues dans chaque case en une seule opération atomique (opération snap()) et également modifier la valeur d'une case (opération write()). Un objet consensus *CNS* supporte une opération unique propose(v) qui retourne une valeur v'. Le type de ces objets se déduit de la spécification du consensus : si v' est retournée alors il existe un processus qui a exécuté CNS.propose(v') et tous les appels qui terminent retournent la même valeur.

La spécification d'un objet impose souvent des restrictions sur son utilisation. Dans une exécution donnée, un objet à *usage unique* ne peut être accédé qu'au plus une

fois par chaque processus. Tous les objets qui correspondent aux problèmes de décision sont à usage unique (*one-shot types*). Le nombre de processus pouvant accéder aux objets est également parfois restreint. De façon plus opérationnelle, un objet dispose de points d'entrées ou *ports* en nombre limité. Un processus invoque une opération sur l'objet en accédant à l'un des ports. Il est généralement supposé que, dans une exécution donnée, un processus accède au plus à un port et que chaque port est accédé par au plus un processus. Une interrogation naturelle est comment ces ports sont reliés aux processus ? Ou, en d'autres termes, quel est le câblage de l'objet ? Un objet câblé *statiquement* (*one-to-one binding scheme* [27]) associe à chaque processus un numéro de port exclusif. L'association est statique dans le sens où le numéro de port est fixé a priori. Par contre, dans le schéma de câblage dynamique, chaque processus peut accéder à n'importe quel port. Un algorithme fondé sur ce type d'objet doit néanmoins assurer que chaque port est associé à au plus un processus et, si l'objet est à usage unique, que chaque processus effectue au plus un seul accès dans toute exécution.

Le comportement d'un objet lorsque plusieurs processus effectuent des opérations concurrentes est déterminé par un critère de cohérence. Les objets que nous étudions sont linéarisables [76] : les opérations semblent s'exécuter de façon instantanée bien qu'elles soient éventuellement concurrentes. De plus, l'ordre des instants auxquels les opérations semblent apparaître est consistant avec leur déroulement dans le temps : si une opération termine avant le démarrage d'une deuxième, la première précède la seconde dans la linéarisation.

Un autre point important à prendre en compte est le comportement des objets lorsque des défaillances surviennent. Nous supposerons que les objets sont *sans attente* [69] : toute opération lancée par un processus correct s'exécute jusqu'à son terme, et ce quelque soit le motif de défaillances. Si un processus qui invoque une opération tombe en panne avant d'avoir reçu la réponse correspondante alors cette opération est susceptible de modifier l'état de l'objet à n'importe quel instant succédant l'invocation.

Réductions lire/écrire sans attente (*wait free read/write reductions [51, 52]*) Maintenant que nous disposons de la notion d'objet, nous définissons le cadre dans lequel nous allons comparer les problèmes. Nous étudions la difficulté relative de différents problèmes dans le modèle à mémoire partagée dans lequel le nombre de défaillances dans chaque exécution est borné par $n-1$.

Soit $P = (\Delta, \mathcal{I}, \mathcal{O})$ un problème de décision. L'hypothèse de l'existence d'une solution sans attente au problème P pour n processus se traduit par un objet O spécifié comme suit. Un tel objet à usage unique exporte une seule opération propose() et dispose de n ports. Pour une exécution donnée, soit I le vecteur des valeurs soumises à l'objet par les invocations propose() ($I[i] = \perp$ si p_i n'exécute pas O.propose()) et J le vecteur des valeurs renvoyées ($J[i] = \perp$ si $I[i] = \perp$ ou p_i défaille avant d'obtenir une réponse de O). O *implémente* P si $I \in \mathcal{I} \Rightarrow \Delta(I, J)$ quelque soit l'exécution considérée.

Soit P' un deuxième problème. Il existe une *réduction sans attente* de P' vers P si il existe un algorithme $\mathcal{A}$ sans attente qui résout P' dans le modèle à mémoire partagée muni d'objets O qui implémentent P. La seule contrainte que $\mathcal{A}$ doit satisfaire est la tolérance à $n-1$ défaillances éventuelles. $\mathcal{A}$ peut utiliser un nombre arbitraire de

registres atomiques et/ou d'objets O. De même, nous ne faisons aucune restriction sur la complexité de $\mathcal{A}$. S'il existe en plus une réduction sans attente de P vers P', nous dirons que les deux problèmes P et P' sont *lire/écrire équivalents sans attente* ou plus simplement *r/w équivalents*.

Notation Nous noterons $\mathcal{SM}_{n,n-1}[O_1, \ldots, O_x]$ le modèles à mémoire partagée composée de n processus. En sus des registres, les processus disposent d'objets $O_1, \ldots, O_x$. Le nombre de registres et d'objets n'est pas borné. Par un léger abus de langage, nous qualifierons parfois ce modèle de sans attente : dans toute exécution, au plus $n-1$ pannes peuvent se produire.

1.4.5 Calculabilité

L'approche « détecteurs de défaillances » vise à caractériser les modèles dans lesquels il existe une solution à certains problèmes fondamentaux. Une autre approche pour étudier de façon systématique la calculabilité consiste à caractériser les problèmes pour lesquels il existe une solution dans un modèle donné. L'étude de cette question a conduit à l'introduction d'outils puissants issus de la topologie dans la théorie du calcul distribué. Nous présentons succinctement dans ce paragraphe les résultats majeurs ainsi que la démarche employée.

Préliminaires L'une des techniques usuelles pour établir des résultats d'impossibilité est la construction d'exécutions indistinguables pour un certain ensemble de processus. En effet, du fait de l'asynchronie du système, un processus ne peut généralement pas à partir de son état local déterminer exactement l'état d'un autre processus. Par exemple, considérons la figure 1.6 qui représente l'état des processus après l'exécution d'une opération write_snap() sur un objet snasphot immédiat IS. Supposons que l'exécution réelle corresponde au triangle central marqué en gras. La figure montre que le processus p_1 ne peut connaître avec précision l'état local des deux autres processus. Ainsi, concernant le processus p_2, la seule certitude de p_1 est que la vue obtenue par p_2 n'est pas $\{1,2\}$. Plus généralement, toutes les exécutions qui correspondent aux triangles dont $(p_1\ , \{1,2,3\})$ est l'un des sommets sont indiscernables de l'exécution courante pour le processus p_1. De même, l'exécution marquée en gras et l'exécution du triangle du milieu dans le coin en bas à droite sont indiscernables à la fois pour p_1 et p_2.

Il s'avère que des structures appelées *simplexes* et *complexes simpliciaux* sont particulièrement adaptées pour décrire des ensembles d'états compatibles et représenter dans quelle « mesure » ces ensembles sont indiscernables.

Nous commençons par donner quelques définitions relatives à ces notions. Un simplexe de *dimension* d ou *d-simplexe* est un ensemble de $d+1$ sommets indépendants. Géométriquement, les sommets sont vus comme des points dans un espace euclidien d'une certaine dimension. Un simplexe est alors l'enveloppe convexe d'un ensemble de points affinement indépendants. Ainsi, un 0-simplexe est un point, un 1-simplexe est représenté par un segment et un 2-simplexe par un triangle plein, etc. Un *complexe simplicial* ou simplement un *complexe* est un ensemble fini de simplexes clos par inclusion

et intersection. La dimension d'un complexe est la plus grande dimension des simplexes qui le composent.

Définition « topologique » d'un problème de décision L'état (ou une partie de cet état) d'un processus peut être représenté par un sommet. Par exemple, dans les figures 1.4 et 1.6, chaque sommet est étiqueté par une paire $s = (v_i, p_i)$ où v_i est la vue obtenue par p_i. Un d-simplexe dont les sommets correspondent à des processus différents représente alors un état global légal, c'est-à-dire composé d'un ensemble d'états locaux compatibles de $d+1$ processus. Par exemple, considérons le consensus binaire pour deux processus. Toutes les configurations de départ sont décrites par le complexe du dessin de gauche dans la figure 1.8. De même, le complexe à droite décrit toutes les configurations finales légales, c'est-à-dire qui respectent la spécification du problème. Ce complexe est formé de deux 1-simplexes disjoints qui dépeignent toutes les décisions possibles. Le simplexe du haut correspond à la situation dans laquelle la décision est 1 tandis que dans celui du bas, tous les processus décident 0. Observons qu'étant donné un simplexe de départ, tous les simplexes finaux ne sont pas légaux : d'après la contrainte de validité du consensus, la décision est nécessairement 0 si tous les processus proposent 0.

Plus généralement, tout problème de décision formulé pour n processus se modélise de manière similaire. Le complexe d'entrée Ic contient un $(n-1)$-simplexe pour chaque vecteur d'entrée possible. De même, le complexe de sortie Oc contient un simplexe pour chaque vecteur de sortie autorisé. Une fonction Δ qui envoie chaque simplexe S de Ic sur un ensemble de simplexes de Oc (dont les sommets sont étiquetés avec les même processus que les sommets de S) définit les sorties légales en fonction des vecteurs d'entrée.

Représenter les algorithmes Comme nous l'avons suggéré, les complexes simpliciaux sont un moyen de décrire si les processus sont capables de différentier plusieurs configurations. Les complexes mesurent à « quel degré deux configurations sont similaires » dans le sens suivant : deux simplexes (i.e., deux configurations) qui partagent un d-simplexe commun sont indiscernables pour au moins d processus.

Soit un algorithme distribué sans attente qui résout un certain problème. Les états finaux possibles des processus sont complètement déterminés par l'ordonnancement de leurs actions et les réponses renvoyées par les objets auxquels ils accèdent. Il est donc possible de définir le *complexe de l'algorithme*. Chaque sommet est étiqueté par l'identité d'un processus et son état à la fin d'une exécution. Étant donné un vecteur d'entrée et un ordonnancement des processus, un simplexe de ce complexe représente l'ensemble des états finaux possibles.

Dans un problème de décision, chaque processus doit choisir une valeur en fonction de son état local final. L'algorithme définit donc une *fonction de décision* δ qui envoie chaque sommet du complexe de l'algorithme vers l'un des sommets du complexe de sortie Oc (étiqueté avec le même processus). Cette fonction est simpliciale[4]. Soit S un simplexe

[4]Soient $C1$ et $C2$ deux complexes et $f : C1 \to C2$. f est simpliciale si pour tout simplex S de $C1$, $f(S)$ est un simplex de $C2$.

du complexe de l'algorithme. Puisque S représente des états finaux compatibles d'un certain ensemble de processus, $\delta(S)$ est nécessairement l'un des simplexes du complexe de sortie Oc représentant un ensemble de décisions compatibles pour ces processus. De plus, δ « respecte » les contraintes de la spécification du problème : si S représente un ensemble d'états atteignables par une exécution à partir du simplexe d'entrée Ic, alors $\delta(S)$ doit être contenu dans $\Delta(Ic)$.

Étudier la calculabilité Nous sommes maintenant prêts à exposer la méthode pour établir des résultats d'impossibilité dans le cadre que nous venons de présenter. Les hypothèses sur le modèle impliquent que les complexes, quelque soit l'algorithme, possèdent certaines propriétés topologiques qui sont préservées par la fonction de décision δ. À partir de la spécification du problème, il est alors possible d'utiliser des théorèmes issus de la topologie pour montrer qu'une telle fonction n'existe pas.

Par exemple, il est possible de montrer que le modèle asynchrone dans lequel les processus accèdent à des objets registres atomiques pour communiquer, tout complexe d'algorithme (qui démarre d'un complexe d'entrée connecté) est connecté [75]. La connectivité est préservée par δ car cette fonction est simpliciale. On peut montrer de cette façon qu'il n'existe pas de solution pour le consensus binaire. Comme dépeint dans la figure 1.8, le complexe d'entrée est connecté. Par contre, le complexe de sortie ne l'est pas. Il s'ensuit que l'image par δ d'un complexe d'algorithme qui résout le consensus est nécessairement déconnectée. Par conséquent, le complexe de l'algorithme est lui aussi déconnecté : une contradiction.

Exemple : $(3,2)$-accord Historiquement, l'approche présentée ci-dessus est née de l'étude des conditions dans lesquelles il existe une solution au problème du consensus ensembliste. Dans trois articles parus à STOC'93, Borowsky and Gafni [21], Herlihhy et Shavit [74], Saks et Zaharoglou [116] ont indépendamment démontré que $k+1$ processus ne peuvent résoudre sans attente le k-accord dans le modèle asynchrone à mémoire partagée.

Pour illustrer cette approche et préparer le terrain en vue du chapitre 4, considérons une forme restreinte du problème du 2-accord pour 3 processus (noté $(3,2)$-accord) dans laquelle la valeur proposée par p_i est toujours i. Nous essayons de donner l'intuition de l'impossibilité de ce problème de manière sans attente dans un système formé de 3 processus. La figure 1.9 dépeint la spécification topologique du problème. Notons la différence de « forme » entre le complexe d'entrée et le complexe de sortie. L'espace des configurations d'entrée valides est représenté par un triangle plein. Par contre, les configurations de sorties légales composent un triangle « vide » : l'intérieur du triangle (à droite sur la figure 1.9) a été enlevé. Enfin, les deux complexes sont connectés : l'observation que le calcul sans attente dans le modèle à mémoire partagée conserve la connectivité ne suffit pas. En fait, la démonstration de l'impossibilité du $(3,2)$-accord se base sur une extension de la notion de connectivité dans les dimensions supérieures. Le calcul sans attente en mémoire partagée a la capacité de déformer l'espace d'entrée, mais ne peut ni y percer des « trous » ni le séparer en plusieurs morceaux.

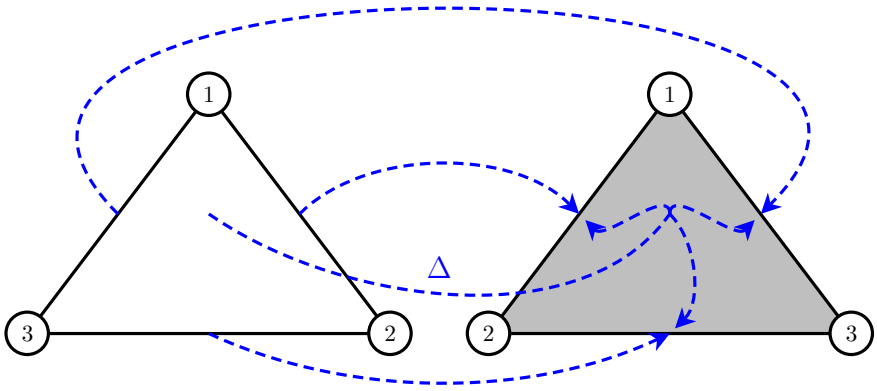

FIG. 1.9 – Spécification « topologique » du $(3,2)$-accord

Suivant Borowsky et Gafni [21, 22, 24], plaçons nous dans un modèle dans lequel les processus communiquent par l'intermédiaire d'objets snapshot immédiat *IS* (cf. paragraphe 1.3.3) au lieu d'accéder directement aux registres de bas niveau. Pour simplifier davantage, les objets *IS* sont à usage unique, organisés en séquence $IS[1], IS[2], \ldots$. Une exécution dans ce modèle (appelé *IIS* pour *Iterated Immediate Snasphot* [24]) consiste simplement, pour chaque processus, à accéder successivement aux objets $IS[1], IS[2], \ldots, IS[B]$ avant décider en fonction de la dernière vue obtenue :

$sm_i \leftarrow input_i$;
for $r = 1$ **to** B **do** $sm_i \leftarrow IS[r]$.write_snap(sm_i) **endfor** ;
$dec_i \leftarrow$ decide(sm_i)

Ce type d'algorithme est dit *totalement informé* (*full information protocol*). À chaque itération ou *ronde* r, un processus poste toute la connaissance qu'il a accumulée sur l'exécution en cours (contenu de la variable sm_i). La vue finale d'un processus p_i est donc totalement déterminée par (1) l'ensemble des entrées et (2) les vues successives renvoyées par les objets *IS*. La dernière vue (obtenue à la B-ième itération) contient la plus grande quantité d'information qu'il est possible à p_i d'accumuler étant donnés les paramètres (1) et (2). Ainsi, dans ce modèle, pour résoudre une tâche de décision, il suffit de déterminer le nombre d'itérations B et concevoir la fonction de décision appropriée.

À quoi ressemble le complexe d'un algorithme dans ce modèle ? La figure 1.10 dépeint les complexes qui correspondent aux algorithmes qui se déroulent en une et deux rondes respectivement. Ces complexes sont hautement structurés. Le complexe qui représente les exécutions comportant $r+1$ rondes s'obtient récursivement à partir du complexe qui correspond à r rondes en subdivisant chaque petit 2-simplexe en suivant le motif du dessin à gauche dans la figure 1.10. Ce procédé ne crée pas de « trou » : le complexe reste plein.

Pour établir l'impossibilité du $(3,2)$-accord dans le modèle *IIS*, supposons par

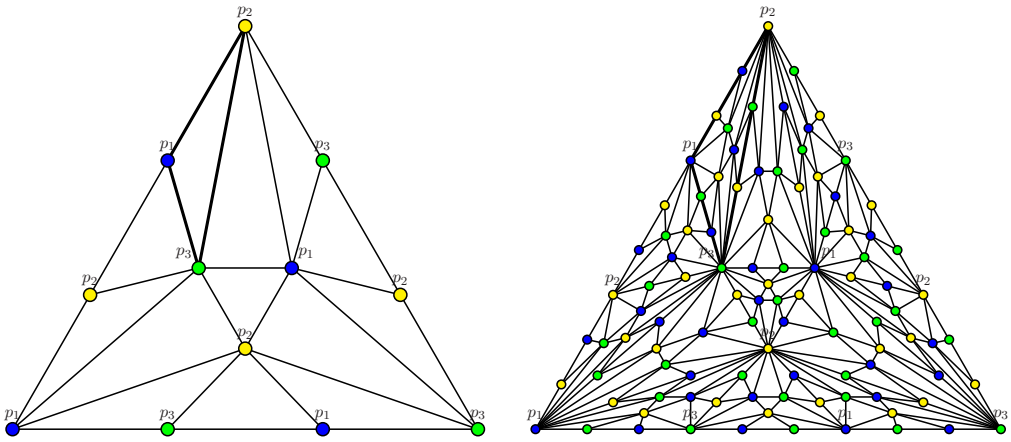

FIG. 1.10 – 1-ronde et 2-ronde complexes

contradiction qu'il existe une fonctions simpliciale δ qui envoie chaque simplexe du r-ronde complexe vers le complexe de sortie associé au $(3,2)$-accord (dessiné à la figure 1.9). De telles fonctions existent : par exemple la fonction triviale qui envoie tout le complexe vers l'un des sommets du complexe de sortie est clairement simpliciale. Le point crucial pour établir le résultat est que δ doit respecter la spécification du problème. Ceci implique que certains simplexes du complexe de l'algorithme ne peuvent être envoyés « n'importe où » dans le complexe de sortie. En particulier, un 1-simplexe sur le bord (du triangle englobant) doit être envoyé sur le bord correspondant du complexe de sortie. Par exemple, un simplexe S sur le bord extérieur gauche des complexes de la figure 1.10 correspond à une exécution dans laquelle seuls p_1 et p_3 participent. Ces processus décident donc nécessairement 1 ou 3. Il s'ensuit que $\delta(S)$ doit être envoyé sur le bord extérieur droit du complexe de sortie (cf. figure 1.9). D'un point de vue topologique, cette contrainte requiert « d'enrouler » une forme pleine (le complexe de l'algorithme) autour d'une forme évidée. L'existence d'une telle opération reviendrait à l'existence d'une déformation continue d'un disque en un anneau, ce qui est impossible.

Plus précisément, dans [21, 75] les démonstrations reposent sur le lemme de Sperner [118]. S'il existe une solution pour le $(3,2)$-accord, alors chaque sommet du complexe de l'algorithme peut être colorié avec la décision $\in \{1,2,3\}$ du processus correspondant. Ce coloriage vérifie les conditions suivantes :

- Les « coins », c'est-à-dire les sommets du triangle englobant sont colorés avec l'identité du processus associé ;
- La couleur d'un sommet appartenant au bord (p_i, p_j) est i ou j ;
- Chaque petit triangle est colorié avec au plus deux couleurs.

Le Lemme de Sperner interdit l'existence d'un tel coloriage.

La démonstration esquissée ci-dessus établit l'impossibilité dans le modèle *IIS*. Dans ce modèle, les entrelacements des lectures et écritures sont plus contraints que dans le modèle à registres atomique. Grossièrement, le modèle *IIS* revient à examiner une

classe restreinte des exécutions du modèle à registre (les exécutions dans lesquelles les opérations successives écriture/lecture semblent instantanées). Il s'avère que les deux modèles sont équivalents du point de vue du calcul sans attente des tâches de décision [24].

Ainsi, du point de vue topologique, les déformations de l'espace d'entrée qu'il est possibles d'effectuer dans le modèle sans attente à registres sont limitées. Que se passe-t-il si l'on augmente le modèle avec des objets plus puissants ? Par exemple, supposons qu'en plus de registres atomiques, les processus communiquent via des objets *test&set* . Quel est l'effet de l'ajout de ces objets sur la puissance de calcul du modèles, notamment sur la possibilité de résoudre le k-accord ?

De même que précédemment, il est possible pour faciliter l'analyse de définir un modèle itéré restreint *IRIS* (*Iterated Restricted Immediate Snapshot*) qui capture la puissance du modèle à registres équipé d'objets test&set. Le complexe des algorithmes de ce modèle possède maintenant des « trous ». L'obstruction topologique qui empêchait l'existence du calcul du $(3,2)$-accord disparaît. La figure 1.11 montre le complexe

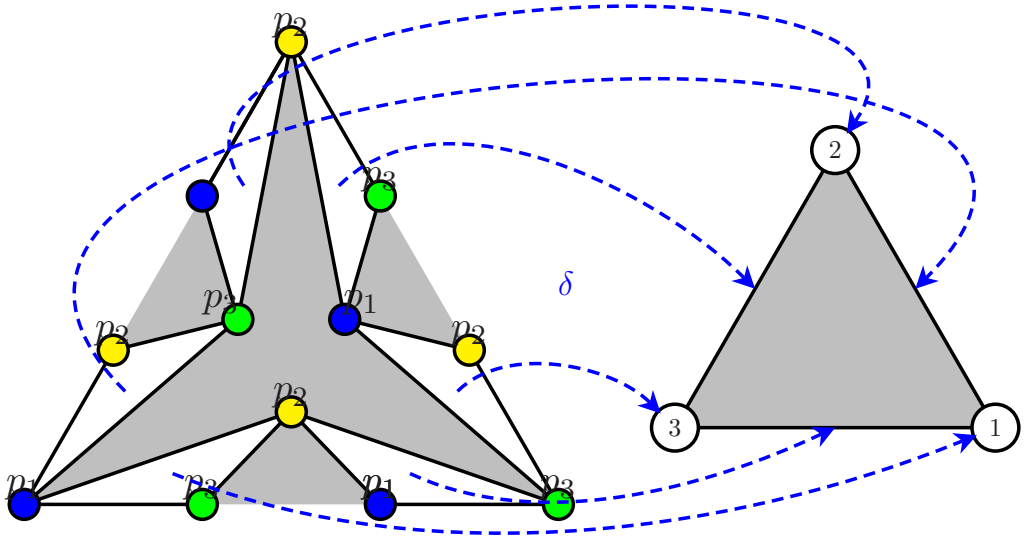

FIG. 1.11 – 1-ronde complexe, modèle à registres + objets test&set

associé à une ronde d'un algorithme totalement informé dans ce modèle $IRIS(test\&set)$. Comparé au complexe de la figure 1.10, certains triangles ont disparu. Le vide laissé crée suffisamment de liberté pour déformer ce complexe vers le complexe de sortie du $(3,2)$-accord. Les flèches en pointillé montrent une déformation possible, c'est-à-dire une fonction simpliciale δ.

Théorème de calculabilité asynchrone L'approche décrite ci-dessus, qui visait initialement à établir l'impossibilité du k-accord a été ensuite développée pour obtenir des résultats plus généraux qui caractérisent les problèmes qui ont une solution sans at-

tente, utilisant uniquement des registres atomiques. Herlihy et Shavit [75] montre qu'un problème de décision peut être résolu si et seulement si il est possible de diviser les simplexes du complexe d'entrée en des simplexes plus petits qui peuvent être envoyés vers le complexe de sortie par une fonction simpliciale μ. Cette fonction μ doit satisfaire des propriétés similaires aux fonctions de décision δ. Notamment, elle doit préserver l'étiquetage des sommets par les noms de processus et respecter la spécification du problème. Si le simplexe s naît d'une divisions du simplexe S du complexe d'entrée alors $\mu(s)$ est un simplexe de $\Delta(S)$. Cette caractérisation est appelée *Asynchronous Computability Theorem* [75]. Ce théorème réduit la question de déterminer si un problème possède une solution à l'étude des propriétés des complexes induits par sa spécification.

Ce résultat a été prouvé de différentes manières, donnant naissance à différentes formulations [73, 19]. En particulier, Borowsky et Gafni [24] donne une preuve dans le modèle snapshot immédiat itéré que nous avons introduit précédemment. Un des points principaux de la démonstration est un algorithme qui simule le modèle classique à registres dans le modèle itéré. C'est cette approche que nous développerons dans le chapitre 4, où nous chercherons à caractériser la puissance fournie par l'addition de détecteurs de défaillances.

Malheureusement, la caractérisation n'est pas effective. Gafni et Koutsoupias [55] ont montré, à partir du théorème de calculabilité asynchrone que déterminer si un problème défini pour trois processus possède une solution sans attente est indécidable. Ce résultat a été étendu par Herlihy et Rajsbaum [72].

Des extensions de l'approche « topologique » ont également été proposées pour prendre en compte des modèles munis d'objets plus puissant que des registres. En particulier, Herlihy et Rajsbaum [73] étudient en détails les propriétés de connectivité des complexes d'algorithmes qui utilisent des objets k-accord.

1.4.6 Difficulté relative des différents problèmes

Nous cherchons à établir les difficultés relatives des familles de problèmes introduits dans le paragraphe 1.4.1. Ce paragraphe présente les principaux résultats connus ayant trait à la classification du point de vue de la calculabilité de différents problèmes de décision.

La capacité de résoudre le consensus est souvent prise comme référence puisqu'il est universel [69] : dans un modèle à mémoire partagée augmenté avec des objets consensus, tout problème qui possède une spécification séquentielle a une solution déterministe. Herlihy a formalisé les différences entre objets par rapport à leur capacité à résoudre le consensus. À chaque objet O correspond un entier c appelé *nombre de consensus* avec la sémantique suivante [69, 77] : un objet O a pour nombre de consensus c si cet objet permet de résoudre le consensus parmi au plus c processus en dépit de jusqu'à $c-1$ défaillances. Soient deux objets O et O' qui ont pour nombre de consensus c et c' respectivement. Supposons que $c < c'$. On déduit de la définition précédente qu'il n'est pas possible d'implémenter un objet O' de façon sans attente dans le modèle à mémoire partagée muni d'objets O (à condition que le système comporte plus de c processus). D'un autre côté, d'après le résultat d'universalité, un objet O possède une

implémentation sans attente fondée sur des registres et des objets O' dans un système composé d'au plus c' processus. Ainsi, ce nombre de consensus classe les problèmes selon leur difficulté respective en une hiérarchie appelée *hiérarchie de Herlihy*.

Par exemple, le nombre de consensus de test&set est 2, celui d'un registre atomique est 1 et *compare&swap* possède un nombre de consensus infini. Cependant, le nombre de consensus ne caractérise pas complètement un problème donné, notamment si l'on s'intéresse à la résolution d'un problème plus faible que le consensus tel le k-accord (voir par exemple [109]).

Dans la même ligne de recherche, Herlihy et Rajsbaum ont étudié les relations [73], à l'aide de méthodes puissantes issues de la topologie algébrique, entre objets (n,k)-accord, c'est-à-dire des objets qui permettent de résoudre de manière sans attente l'accord sur k valeurs parmi n processus. Ils établissent des conditions nécessaires et suffisantes pour l'existence d'une solution au (n,k)-accord à partir d'objets (n',k')-accord et de registres atomiques.

1.5 Résultats

Ce document présente différents résultats ayant trait à la difficulté relative des problèmes de coordination faible, et investigue les détecteurs de défaillances introduits dans la littérature dans le but de résoudre ces problèmes. Nous présentons une partie des travaux qui ont fait l'objet des publications [4, 39, 58, 59, 92, 93, 98, 100]. Certains de ces résultats ont été obtenus en collaboration avec S. Rajsbaum, E. Gafni et/ou A. Mostéfaoui. Les travaux publiés dans [49, 97, 102, 99, 101, 110, 111, 113, 114] ne seront pas discutés dans ce document.

Renommage et accord La difficulté du renommage dépend du cardinal de l'espace des nouveaux noms. Plus l'espace est vaste, moins le problème est difficile. Le cardinal de l'espace de renommage est défini par une fonction h qui détermine pour un nombre de processus participant p l'intervalle $[1,\ldots,h(p)]$ dans lequel ces p processus doivent acquérir un nouveau nom.

Nous cherchons à mieux comprendre l'influence de la « qualité » de l'accord sur l'espace minimal de renommage. Quelle est, en utilisant des registres et des objets (n,k)-accord ou (n,k)-test&set, le plus petit espace renommage qu'il est possible d'atteindre ?

Nous montrons qu'à l'aide d'objets (n,k)-test&set, il est possible de construire un objet renommage dont l'espace est déterminé par une certaine fonction f_k ($f_k : p \rightarrow 2p - \lceil \frac{p}{k} \rceil$). Cet espace est optimal : réciproquement nous montrons qu'à partir d'un objet renommage qui garantit cet espace, on peut construire un objet (n,k)-test&set. Ces résultats sont parus dans [98].

Dans une communication postérieure à la publication de [98], Gafni a présenté un algorithme de renommage optimal (en terme de cardinal de l'espace des nouveaux noms) fondé sur le (n,k)-accord [53]. Le cardinal de l'espace est défini par la fonction h_k : $p \rightarrow \min(2p-1, p+k-1)$. Réciproquement, nous montrons dans [100] que pour $k = t$, renommage en $h_t(p)$ noms et (n,t)-accord sont t-résilient équivalents en mémoire

partagée. De plus, [100] établit que dans le cas général, renommage en $h_k(p)$ noms est strictement plus facile que le (n,k)-accord.

Accord ensembliste et accord sur plusieurs fronts Nous avons introduit ce nouveau schéma de coordination dans [58], à la suite des travaux de Gafni et Rajsbaum [57]. Le problème (n,k)-comité abstrait l'idée de mener simultanément plusieurs protocoles de consensus. Une instance du problème du consensus est appelée comité. Les n processus essaient de résoudre k instances en parallèle, avec la contrainte de décider dans au moins un comité. Les décisions au sein d'un même comité doivent être les mêmes. Nous nous demandons naturellement comment ce problème se compare au (n,k)-accord. Dans [58], nous montrons en combinant outils issus de la topologie et réductions algorithmiques que les deux problèmes sont équivalents, même si on se restreint à des valeurs binaires en entrée des comités. Les résultats sont exposés dans le cadre de 3 processus et pour $k=2$. Ils se généralisent pour tout n à $k=n-1$. Dans [4] nous étendons le résultat à tout n et tout k à l'aide de techniques purement algorithmiques. La version binaire du problème décision de comité peut être interprété comme le pendant « binaire » de l'accord ensembliste.

Composition de détecteurs de défaillances Tout les problèmes cités ci-dessus sont non triviaux : il n'existe pas d'algorithme sans attente pour les résoudre dans un environnement asynchrone. Plusieurs détecteurs de défaillances ont été proposé dans la littérature pour contourner ces impossibilités. Dans [92], nous comparons les puissances de certaines familles de détecteurs orientés vers la résolution du (n,k)-accord. Pour chaque paire $(\mathcal{C}1,\mathcal{C}2)$ dans ces familles, nous caractérisons les conditions dans lesquelles il existe une réduction de $\mathcal{C}1$ vers $\mathcal{C}2$ ou de $\mathcal{C}2$ vers $\mathcal{C}1$. Nous classons également ces détecteurs par rapport à leur capacité à résoudre le (n,k)-accord. Le rang d'un détecteur $\mathcal{C}$ dans cette hiérarchie est le plus entier k tel qu'il existe un algorithme fondé sur $\mathcal{C}$ qui résout le (n,k)-accord. Finalement, nous établissons que cette hiérarchie n'est pas robuste : il existe des détecteurs $\mathcal{C}1$ et $\mathcal{C}2$ de rang respectif $k1$ et $k2$ dans la hiérarchie que l'on combine pour obtenir un détecteur de rang $k3$ avec $k3 < \min(k1,k2)$. Ce résultat suggère que l'espace entre modèle purement synchrone et purement asynchrone n'est pas linéaire. Dans [93], nous spécialisons les techniques algorithmiques développées dans [92] pour donner une transformation extrêmement simple du détecteur $\diamond\mathcal{W}$ vers le détecteur Ω, offrant ainsi une alternative aux transformations plus complexes de Chandra et coauteurs [29] et Chu [37].

Autres résultats (qui ne seront pas discutés dans ce document) En plus des résultats présentés dans ce document, j'ai été concurremment ou précédemment impliqué dans plusieurs activités de recherche, qui ont donné lieu à un certain nombre de publications résumées ici.

- Compléxité de l'accord ensembliste en environnement synchrone, sous différentes modèle de défaillances. Nous avons construit des algorithme optimaux (en nombre de rondes) qui résolvent le k-accord dans des modèles synchrones sujets aux dé-

faillances par omission [111, 110, 114]. Ces algorithmes permettent aux processus de décider au plus tôt, c'est-à-dire dès que les conditions de l'exécution permettent une décision sûre.
- Abstractions pour systèmes dynamiques. Dans [102, 49] nous proposons différentes abstractions simples adaptées aux système dynamiques, notamment pour la construction de registres (quorums) ou la diffusion d'informations avec certaines garanties de fiabilité.
- Mise en œuvre des détecteurs de défaillances [97, 88, 99]. Étant donné un détecteur de défaillances, cette ligne de recherche vise à identifier les hypothèses physiques minimales (par exemple, l'existence de propriétés synchrones sur certains canaux de communication) qui, lorsqu'elles sont satisfaites par le système, permettent de construire le détecteur.

1.5.1 Organisation du document

Le reste du document se divise en 3 chapitres.
- Le chapitre 2 présente les différentes réductions lire/écrire entre les problèmes de coordination faibles. Nous introduisons d'abord les spécifications précises des problèmes qui nous intéressent (paragraphe 2.1). Le paragraphe 2.3 présente une série d'algorithmes qui, une fois rassemblés établissent l'équivalent entre renommage et test&set ensembliste. Des résultats d'impossibilité et des extension au calcul t-résilient ont été publiés dans [100]. Le paragraphe 2.4 établit que (n, k)-accord et (n, k)-comité binaires sont r/w-équivalents sans attente.
- Le chapitre 3 se consacre à l'étude de familles de détecteurs pour résoudre le consensus ensembliste. L'organisation de ce chapitre suit le plan du chapitre précédent. Les paragraphes 3.1 et 3.2 spécifient les familles de détecteurs étudiées. La capacité de ces détecteurs à résoudre le (n, k)-accord est discutée dans le paragraphe 3.3. Nous présentons l'algorithme d'addition dans le paragraphe 3.4. Finalement, nous établissons dans le paragraphe 3.5 des conditions nécessaires et suffisantes pour l'addition et/ou la réduction pour chaque paire de détecteurs des différentes familles étudiées.
- Enfin, dans le dernier chapitre, nous tentons de définir un modèle itéré, similaire au modèle *IIS* (brièvement introduit dans le paragraphe 1.4.5) qui capture la puissance de calcul additionnelle offerte par différents détecteurs de défaillances. L'approche, et son intérêt éventuel, est discuté dans le paragraphe 4.1. Le paragraphe 4.2, pour chacun des détecteurs du chapitre 3, montre comment construire un modèle itéré restreint. Réciproquement, le paragraphe 4.3 présentent les transformations inverses : comment, dans le modèle itéré restreint induit par un détecteur $\mathcal{C}$, extraire le détecteur $\mathcal{C}$ associé. Le théorème introduit dans le paragraphe 4.4 montre que le modèle itéré conserve la puissance du détecteur initial si l'on se restreint à une certaine classe de problème. Enfin, ce formalisme est exploité dans le paragraphe 4.5 où nous établissons par des preuves simples des conditions nécessaires et suffisantes pour résoudre la résolution du (n, k)-accord à l'aide de certains détecteurs.

Notations

$\mathbb{N}$	l'ensemble des entiers naturels
$\mathbb{T}$	instants de l'horloge globale
n	nombre de processus
p_i	le processus d'index i
$id(p_i) \in \mathbb{N}$	l'identité de p_i
t	nombre maximal de défaillances par panne franche
f	nombre effectifs de processus défaillants
$\tau \in \mathbb{T}$	un instant
x_i	variable locale du p_i
x_i^τ	valeur de x_i à l'instant τ
$\Pi = \{p_1, \ldots, p_n\}$	l'ensemble des processus du système
	désigne aussi l'ensemble des index des processus
R	registre atomique
$R[1..n]$	tableau de registre atomique
OBJ	objet partagé
VAR_i	variable locale accessible en lecture seule
$\mathcal{C}$	classe de détecteurs de défaillances
$\mathcal{MP}_{n,t}[\emptyset]$	modèle asynchrone à passage de message
	n : nombre de processus
	t : borne supérieure sur le nombre de processus défaillants
$\mathcal{MP}_{n,t}[\mathcal{C}, OBJ]$	$\mathcal{MP}_{n,t}[\emptyset]$ enrichi avec un détecteurs de la classe $\mathcal{C}$
	et une infinité d'objets OBJ
$\mathcal{SM}_{n,t}[\emptyset]$	Modèle asynchrone à mémoire partagée
$\mathcal{MP}_{n,t}[\mathcal{C}, OBJ]$	$\mathcal{SM}_{n,t}[\emptyset]$ équipé d'un détecteur de la classe $\mathcal{C}$
	et d'une infinité d'objets OBJ
primitive()	primitive offerte par le modèle sous-jacent
send()/receive()	communication point à point (modèle $\mathcal{MP}_{n,t}[\emptyset]$)
broadcast()/deliver()	primitives de la diffusion (modèle $\mathcal{MP}_{n,t}[\emptyset]$)
R_bcast()/R_deliver()	primitives de la diffusion fiable (modèle $\mathcal{MP}_{n,t}[\emptyset]$)
write()/collect()	primitives de la collecte (modèle $\mathcal{SM}_{n,t}[\emptyset]$)
write()/snap()	primitives de la collecte ordonnée (modèle $\mathcal{SM}_{n,t}[\emptyset]$)
write_snap()	primitive de la collecte ordonnée immédiate (modèle $\mathcal{SM}_{n,t}[\emptyset]$)

Chapitre 2

Relations entre problèmes de coordination faible

Dans ce chapitre, nous étudions du point de vue de la calculabilité les relations entre plusieurs problèmes de coordination faible. Les problèmes que nous étudions sont des abstractions de schémas de coordination courants dans le monde distribué. Ainsi le *renommage* est une instance du problème plus général de l'allocation de ressources. Le k-accord rencontré dans le chapitre 1 est une généralisation du problème de l'élection. L'abstraction *k-test&set* généralise la primitive de synchronisation classique test&set. Enfin, le problème décision de comité, introduit dans [58], est un schéma de coordination nouveau. Les spécifications précises sont données dans la section 2.1. Nous établissons des liens entre ces problèmes via des réductions algorithmiques sans attente dans le modèle asynchrone à mémoire partagée avec défaillances des processus.

Ces problèmes sont non-triviaux : il n'existe pas d'algorithmes pour les résoudre dans le modèle asynchrone, avec possibilité de défaillances. De plus, les démonstrations connues de ces résultats sont fondées sur l'utilisation d'outils sophistiqués, issus de la topologie algébrique. Le théorème de calculabilité asynchrone [75] donne une caractérisation des tâches de décision que l'on peut résoudre en environnement distribué, à l'aide de registres atomiques. Cependant, cette caractérisation n'est pas effective. Chaque nouveau problème nécessite donc le développement d'algorithmes adaptés (exhiber une solution) ou une nouvelle preuve d'impossibilité. D'autre part, adapter ce théorème à des modèles munis d'objets plus puissants que des registres n'est pas une tâche aisée [73, 33].

Les résultats de ce chapitre s'inscrivent dans la lignée des travaux d' E. Gafni, et plus généralement contribuent à enrichir la boîte à outils algorithmiques du distribué. Face à un problème donné, compte tenu de la difficulté de prouver directement l'impossibilité de son calcul, il est préférable d'essayer de découvrir une réduction d'un problème réputé impossible vers la tâche étudiée. De plus, les réductions présentées font apparaître une certaine unité entre ces schémas de coordination, bien qu'ils soient en apparence de natures complètement différentes. Cette unité suggère l'existence d'une petite classe de problèmes, impossibles à calculer, à laquelle tout problème qui n'a pas de solution sans

attente pourrait être réduit.

2.1 Coordination faible

Ce paragraphe présente les différentes familles de problèmes étudiés dans ce chapitre. La spécification de chaque problème possède deux paramètres n et α. Le paramètre n désigne le nombre maximal de processus participant. Le paramètre α représente « le degré de coordination » du problème. Il s'agit le plus souvent d'un entier. Par exemple, il définit le nombre de consensus simultanés dans le (n,k)-comité ou le nombre maximal de valeurs qui sont décidées dans le (n,k)-accord. Pour le renommage α est une fonction du nombre de participants qui spécifie le cardinal maximal de l'espace de renommage. Ainsi, plus α est « petit », plus le problème est contraint et difficile.

2.1.1 Consensus ensembliste

L'accord ensembliste ou *(n,k)-accord* relâche la propriété d'accord du consensus en autorisant la décision d'au plus k valeurs distinctes. Ce problème a été introduit par Chaudhuri [31, 32] dans le but d'étudier du point de vue de la calculabilité asynchrone la relation entre qualité de l'accord (le paramètre k) et degré de tolérance aux défaillances (le paramètre t qui, dans un modèle, indique le nombre maximal de défaillances dans chaque exécution).

Soit v_i la valeur proposée par p_i et d_i la valeur qu'il décide ($d_i = \perp$ si p_i ne décide pas). Toute implémentation valide du (n,k)-accord doit satisfaire les propriétés suivantes pour toute exécution dans laquelle le nombre de participants est borné par n.

Définition 2.1 ((n,k)-accord)
Validité *: $d_i \neq \perp \Rightarrow \exists p_j$ processus participant tel que $d_i = v_j$;*
Terminaison *: p_i participe et est correct $\Rightarrow d_i \neq \perp$;*
Accord *: $|\{d_i : d_i \neq \perp\}| \leq k$.*

L'accord ensembliste peut également être vu comme une généralisation de l'un des problèmes fondamentaux du calcul réparti : l'élection [121, 82]. De la même façon que le consensus et l'élection (les processus se mettent d'accord sur l'identité d'un des participants) sont deux problèmes équivalents dans le contexte du calcul tolérant aux défaillances, l'accord ensembliste peut être vu comme une généralisation de l'élection[21, 52]. Le problème de l'élection multiple ou *k-élection* consiste pour chaque processus à choisir un leader de telle sorte que le nombre final de leaders n'excède pas k. De plus, chaque leader doit se percevoir lui même comme un leader. Il n'est pas difficile de résoudre le (n,k)-accord à partir d'une solution à ce problème (chaque processus décide la valeur proposée par son leader). En revanche, la transformation inverse est un peu plus complexe. En effet, un objet qui implémente le (n,k)-accord ne garantit pas que chaque valeur choisie soit décidée par le processus qui la propose. Une solution sans attente en mémoire partagée est donnée dans [21].

Calculabilité L'existence d'un algorithme pour résoudre le (n,k)-accord dans le modèle asynchrone $\mathcal{SM}_{n,t}[\emptyset]$ ([1]) dépend des paramètres n, k et t. Lorsque $k > t$, il existe une solution triviale (code pour p_i) :

1. Écrire v_i (valeur proposée) ;
2. Lire la mémoire partagée jusqu'à observer au moins $n-t$ propositions de processus différents ;
3. Décider la plus petite valeur observée.

Soit V l'ensemble des k plus petites valeurs proposées. Comme $|V| = k > t$, chaque processus qui décide observe au moins l'une de ces valeurs. L'ensemble des valeurs décidées est donc contenu dans V. Au contraire, si $k \leq t$, il n'existe pas d'algorithme déterministe qui résout le (n,k)-accord [21, 75, 115].

2.1.2 Accord sur plusieurs fronts ou décision de comité

En pratique, on utilise le consensus dans la réplication active. Les processus exécutent une séquence d'opérations et doivent s'accorder sur le résultat de chaque opération avant de traiter la suivante. Cette technique garantit que les états successifs des processus sont les mêmes et permet donc de tolérer la défaillance d'un ou plusieurs processus. Dans ce cas, l'utilisation du consensus est séquentielle : un processus propose une valeur et décide dans une instance du consensus avant de faire appel à l'instance suivante pour proposer et décider une autre valeur.

Il semble naturel d'envisager que les processus proposent et cherchent à décider simultanément dans plusieurs instances du consensus. On peut imaginer que pour assurer le progrès global du calcul, il est suffisant pour un processus d'obtenir une décision dans l'une des instances. Par exemple, si la technique de réplication active est utilisée pour k applications menées en parallèle dans le système, ce schéma garantit en dépit des défaillances éventuelles, le progrès d'au moins l'une d'entre elles. Dans [58], nous introduisons le problème de la *décision de comité* qui formalise ce schéma de coordination.

Dans le problème *(n,k)-comité*, chaque processus p_i propose un vecteur V_i de k valeurs, chaque valeur étant destinée à l'une des k instances du problème du consensus appelée *comité*. Chaque processus doit décider un couple (numéro de comité, valeur) tel que les décisions dans un même comité respectent la spécification du consensus. Nous notons $V_i \in \mathcal{V}^k$ le vecteur proposé par p_i et $(c_i, d_i), (1 \leq c_i \leq k)$ le couple décidée par p_i ($(c_i, d_i) = (\perp, \perp)$ si p_i ne décide pas). Une implémentation valide du (n,k)-comité doit satisfaire la spécification suivante :

Définition 2.2 ((n,k)-comité)
Validité $(c_i, d_i) \neq (\perp, \perp) \Rightarrow \exists p_j$ *processus participant tel que* $V_j[c_i] = d_i$;
Terminaison $\forall p_i$ (p_i *processus correct et participant*) $\Rightarrow (c_i, d_i) \neq (\perp, \perp)$;
Accord $\forall i, j : (c_i = c_j) \Rightarrow (d_i = d_j)$.

[1] Rappelons que cette notation désigne le modèle à registres atomiques, formé de n processus. Dans chaque exécution de ce modèle, au plus $t < n$ d'entre eux sont susceptibles de défaillir.

Remarquons que, comme le (n,k)-accord, le (n,k)-comité est le problème du consensus lorsque $k=1$. Il existe également une solution triviale lorsque $k>t$. Pour simplifier, on suppose que les n processus sont identifiés par les entiers $1,\ldots,n$. Chaque processus p_i exécute les actions suivantes :

1. Si $1 \leq i \leq t+1$ alors écrire V_i dans la mémoire partagée.
2. Lire la mémoire jusqu'à observer un vecteur V_j déposé par un processus p_j tel que $1 \leq j \leq t+1$.
3. Décider $(j, V_j[j])$.

Nous observons également que dans un système équipé d'un objet qui implémente le (n,k)-comité, il est facile de résoudre le (n,k)-accord et ce quelle que soit la valeur du paramètre t. Chaque processus p_i propose au (n,k)-comité le vecteur $V_i = [v_i,\ldots,v_i]$ où v_i est la valeur proposée par p_i . En retour, p_i obtient une paire (b,v') et décide v'. Le nombre de valeurs décidées est bornée par k car il y a k comités et au plus une valeur est décidée dans chaque comité. Ce premier exemple de réduction s'appuie sur l'hypothèse implicite que l'espace $\mathcal{V}$ des valeurs proposables est le même dans les deux problèmes. Afin d'obtenir une compréhension plus fine de ces schémas de coordination, nous nous concentrons sur le problème *décision de comité binaire* dans lequel les valeurs d'entrée appartiennent à $\{0,1\}$, i.e., chaque vecteur V_i appartient à $\{0,1\}^k$.

Définition 2.3 ((n,k)-comité-binaire) *Chacun des n processus p_i propose un vecteur $V_i \in \{0,1\}^k$ et décide un couple (c_i,d_i) qui satisfait les contraintes de terminaison, de validité et d'accord du (n,k)-comité (cf. définition 2.2).*

La notion d'accord sur plusieurs fronts a été initialement formalisée par Gafni et Rajsbaum dans [57]. Le problème des *bancs musicaux* (*musical benches*) [57] est défini par deux paramètres n et k. Le paramètre n désigne le nombre maximal de participants tandis que le paramètre k désigne le nombre de bancs. Chaque banc b_j possède deux places que nous noterons j et $-j$. Initialement, les processus sont arbitrairement « assis » sur les bancs. Le problème consiste à assigner à chaque processus une place de telle sorte que sur chaque banc au plus une position soit occupée. De plus, lorsque la configuration initiale est exempte de conflits (sur chaque banc, au plus une place est occupée), les processus ont interdiction de se déplacer. En résumé, si on note pos_i la position initiale de p_i, les positions finales d_i doivent satisfaire les contraintes suivantes :

1. $(\forall i,j : (|pos_i| = |pos_j| \Rightarrow pos_i = pos_j)) \Rightarrow (\forall i : d_i = pos_i)$ (pas de changements de place en l'absence de conflits dans la configuration initiale) ;
2. $\forall i,j : (|d_i| = |d_j|) \Rightarrow (d_i = d_j)$ (au plus une place occupée par banc dans la configuration finale).

Le rapport avec la notion d'accord sur plusieurs fronts apparaît si l'on considère chaque banc comme une instance du consensus binaire. En effet, lorsque la configuration initiale présente des conflits, les processus participants doivent résoudre au moins l'une de ces instances pour atteindre une configuration sans conflit.

Pour étudier la calculabilité, il est courant de se concentrer sur un certain ensemble de configurations initiales. Une façon de faire cela est de spécifier une variante équivalente P' du problème originale P définie pour $n' > n$ processus. Le problème P' n'a pas d'entrées mais des règles qui décrivent les ensembles possibles de participants. La participation du processus p'_α signifie pour le problème P la participation d'un certain processus p_i proposant une certaine valeur p_i.

Ainsi, pour $n = 3$ (au plus 3 participants) et $k = 2$ (2 bancs), les positions sur chaque banc sont notés $-1, 1$ et $-2, 2$, respectivement. Le système comporte 4 processus p_1, p_{-1}, p_2 et p_{-2} ; p_2 et p_{-2} ne pouvant participer ensemble. Lorsqu'il participe, le processus p_i est initialement assis sur le banc $|i|$ à la position i. La relation Δ (paragraphe 1.4.2) qui spécifie les positions autorisées en sortie en fonction des positions initiales est décrite dans la Figure 2.1 (certaines arêtes sont omises pour éviter de surcharger le dessin).

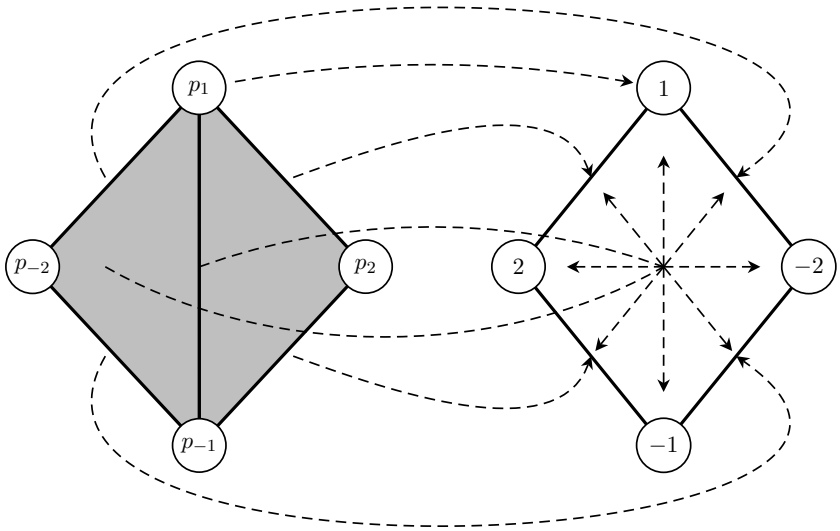

FIG. 2.1 – Spécification des bancs musicaux ($n = 3, k = 2$)

[57] établit qu'il n'est pas possible de résoudre les bancs musicaux sans attente dans le modèle $\mathcal{SM}_{n,n-1}[\emptyset]$. La preuve de l'impossibilité exhibe une obstruction topologique fondée sur le théorème de Borsuk-Ulman [84]. Dans le cas $n = 3, k = 2$, [57] montre également comment résoudre les bancs musicaux à l'aide d'objets $(3, 2)$-accord. La question de l'équivalence entre les problèmes « accord sur plusieurs fronts » et accord ensembliste est approfondie dans [58] où nous montrons que bancs musicaux, (n, k)-accord et (n, k)-comité-binaire sont équivalents pour $n = 3$ et $k = 2$. Les techniques utilisées combinent réductions algorithmiques et la vision du calcul distribué comme déformation d'espaces topologiques. Les méthodes développées se généralisent au cas $k = n - 1$. La recherche d'un résultat d'équivalence ou d'impossibilité pour k quelconque ($1 < k < n - 1$) nous a conduit vers une approche purement algorithmique [4]. C'est cette dernière approche que nous présentons dans ce document (paragraphe 2.4).

2.1.3 Test&Set ensembliste

Dans ce paragraphe, nous complétons le panorama des problèmes d'accord faible en donnant une spécification affaiblie de la primitive test&set().

Cette primitive de synchronisation classique permet de lire et de modifier en une seule opération atomique une variable partagée. Considérons un bit partagé, initialisé à 1, accédé au plus une fois par chaque processus par appel de la primitive test&set(). La primitive test&set() met le bit à 0 et retourne la valeur précédente du bit. Ainsi, dans toute exécution, un seul processus retourne 1 (le *gagnant*) et l'appel retourne 0 pour chacun des autres processus (ces processus sont dits *perdants*). Ce problème est le $(n, 1)$-test&set.

De la même manière que l'accord ensembliste généralise le problème du consensus, le problème *(n, k)-test&set* est une simple généralisation du $(n, 1)$-test&set : *au moins un et au plus k* processus sont gagnants. Ce problème de décision n'a pas d'entrée. Les processus décident en sortie 0 ou 1 en respectant les contraintes suivantes. Nous notons $d_i \in \{0, 1\}$ la valeur décidée par le processus p_i ($d_i = \perp$ si p_i ne décide pas).

Définition 2.4 ((n, k)-test&set)
Terminaison $\forall i : p_i$ *participe et est correct* $\Rightarrow d_i \in \{0, 1\}$;
Accord $1 \leq |\{i : d_i = 1\}| \leq k$.

Dans la hiérarchie de Herlihy [69] (voir aussi le chapitre 1), un objet qui implémente le problème du $(n, 1)$-test&set est au niveau deux. C'est-à-dire qu'il existe un algorithme sans attente de consensus dans un système constitué de deux processus muni d'un objet $(2, 1)$-test&set. De façon similaire, les problèmes $(k + 1, k)$-test&set et $(k + 1, k)$-accord sont équivalents [52, 21]. Nous présentons dans le paragraphe 2.3.1 un algorithme sans attente principalement dû à Gafni qui implémente un objet (n, k)-test&set à partir d'objets $(k+1, k)$-test&set et de registres atomiques. La construction est ensuite utilisée pour montrer l'équivalence entre $(k+1, k)$-test&set et renommage adaptatif. Ce dernier problème est défini dans le paragraphe suivant.

2.1.4 Renommage Adaptatif

Dans le problème du renommage adaptif, chaque processus p_i est initialement muni d'un nom id_i appartenant à l'intervalle $[1 \ldots, N]$. Il s'agit de concevoir un algorithme qui fournit à chaque processus un nouveau nom dans un intervalle plus petit $[1, \ldots, M]$ avec $M \ll N$. De plus, on souhaite que la taille de l'espace de renommage dépende du nombre de processus qui participent. Le paramètre principal du problème est donc la fonction $h : \mathbb{N} \rightarrow \mathbb{N}$ qui définit la taille de l'espace de renommage à partir du nombre participants. Le second n indique le nombre maximal de participants. Les p processus participant doivent acquérir des noms distincts dans l'intervalle $[1 \ldots, h(p)]$. Formellement,

Définition 2.5 ((n, h)-renaming) *Le nouveau nom acquis par p_i est noté new_ id_i et p ($1 \leq p \leq n$) désigne le nombre de processus participant.*
Terminaison *Tout processus correct qui participe obtient un nouveau nom ;*

Unicité $\forall i, j : i \neq j \Rightarrow new\_id_i \neq new\_id_j$;
Validité $\forall i : new\_id_i \in [1, \ldots, h(p)]$.

Pour éviter les solutions triviales, il est usuellement requis que toute solution satisfasse une certaine forme d'anonymat par rapport aux index i des processus. Bien que l'algorithme trivial dans lequel chaque processus p_i choisit comme nouveau nom son index i ne soit pas adaptatif, on peut penser que l'utilisation de l'index des processus facilite la conception d'algorithmes de renommage. Une contrainte qui empêche ce type de solution est usuellement formulée ainsi [121] :

– Le code exécuté par le processus p_i dans une exécution où son nom initial est id est exactement le même que le code exécuté par p_j initialement nommé id.

Autrement dit, les processus se distinguent uniquement par leurs noms initiaux que l'on supposera toujours totalement ordonnés.

Ce problème a été proposé et résolu par Attiya et coauteurs [15] dans le modèle à passage de messages. Ils montrent que $t < \frac{n}{2}$ est une condition nécessaire pour résoudre le problème du renommage dans le modèle $\mathcal{MP}_{n,t}[\emptyset]$. Ils présentent également un algorithme dont l'espace de renommage est $n+t$. Bar-Noy et Dolev [20] donnent une solution dans le modèle à mémoire partagée. Burns et Peterson [26] étudient le problème de l'allocation dynamique de ℓ ressources distinctes (ℓ-*assignment*). Ils présentent un algorithme sans attente qui suppose $\ell \geq 2k-1$, où k est le nombre de processus qui souhaitent acquérir une ressource. Quel est le plus petit espace de renommage qu'il est possible d'atteindre de façon sans attente dans un modèle à registres ? Il a été montré que $M = 2p-1$ est la borne inférieure dans un tel contexte [75]. La démonstration repose sur le théorème de calculabilité asynchrone.

Notre étude porte sur des problèmes non triviaux, c'est-à-dire des problèmes qui n'ont pas de solution dans le modèle $\mathcal{SM}_{n,n-1}[\emptyset]$. Par conséquent, nous nous intéressons aux problèmes de renommage qui requièrent que la taille de l'espace des nouveaux noms soit sous la frontière $p \rightarrow 2p-1$. En particulier, nous nous concentrons sur les deux familles de fonctions $(f_k)_{1\leq k\leq n}$ et $(g_k)_{1\leq k\leq n}$ qui définissent l'espace de renommage (voir Figure 2.2) :

$$\begin{aligned} f_k : \mathbb{N} &\rightarrow \mathbb{N} \\ p &\rightarrow 2p - \lceil \frac{p}{k} \rceil \\ g_k : \mathbb{N} &\rightarrow \mathbb{N} \\ p &\rightarrow \min(2p-1, p+k-1) \end{aligned}$$

Le cas $k = 1$ correspond au renommage parfait (i.e., f_1 et g_1 sont égales à la fonction identité ; p processus se renomment dans l'espace $[1, p]$). À l'autre extrême ($k = n$), nous retombons sur le plus petit espace qu'il est possible d'obtenir de façon sans attente en utilisant uniquement des registres (i.e., $f_n = g_n = p \rightarrow 2p-1$).

À la différence du (n,k)-accord dont, lorsqu'elle existe, la solution est triviale, l'algorithmique du renommage adaptif est très riche [5, 16, 17, 121, 22, 86, 107]. Il existe de nombreux travaux sur le sujet qui étudient principalement le compromis entre taille de l'espace de renommage et complexité des algorithmes. En effet, le renommage adaptatif

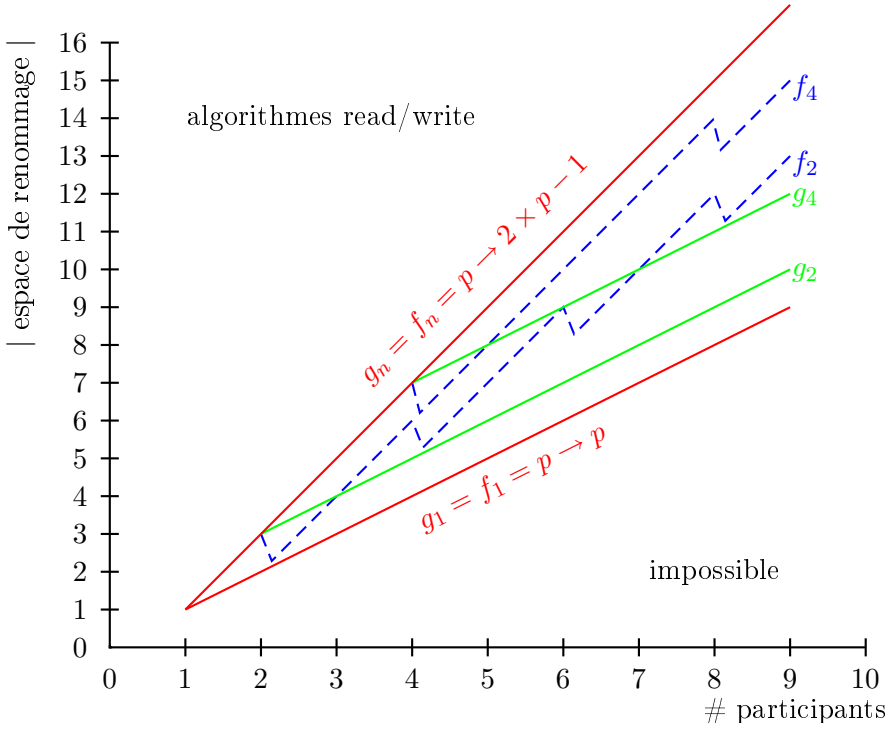

FIG. 2.2 – Espaces de renommage

est une brique de base dans la conception d'algorithmes dont la complexité dépend du nombre effectif de participants et non du nombre total de processus. L'utilisation du renommage adaptatif pour améliorer les performances des algorithmes dont la complexité dépend de la taille de l'espace initial des noms des processus a été suggérée par Anderson et Moir [12]. Le nouveau nom obtenu grâce au renommage remplace le nom original dans l'algorithme initial et la complexité dépend alors du nombre de processus actifs. Cette stratégie est employée dans plusieurs algorithmes dont la complexité dépend du nombre de participants [6, 7, 86, 17].

Le problème du renommage peut également être vu comme une variante du problème classique de l'allocation de ressource. Le système comporte un certain nombre de ressources $r_1, \ldots, r_\alpha$ identiques, et, pour mener a bien son calcul, chaque processus doit acquérir l'accès exclusif à une de ces ressources. Dans ce type de scénario, il est naturel de supposer que les ressources ne sont utilisées que pour une durée finie. Dans le problème du *renommage continu* (*long-lived renaming* [12, 121]) les processus acquièrent et relâchent les noms à l'aide des primitives get_name(*id*) et release(*name*). Notre étude se limite au renommage à usage unique (*one-shot*).

2.1.5 Participating set

Le *participating set* [22] est la formulation orientée « problème » de l'abstraction collecte ordonnée immédiate (snapshot immédiat, cf. paragraphe 1.3.3). Chaque pro-

cessus p_i propose son identité id_i et doit produire en sortie une vue S_i des processus participants (c'est-à-dire un ensemble d'identités). Les vues doivent satisfaire les propriétés d' auto-inclusion, de comparaison et de simultanéité de la collecte ordonnée. Le (n,k)-participating-set ajoute une contrainte supplémentaire sur le nombre de processus qui obtiennent la même vue : pour toute vue S, le cardinal de l'ensemble des processus qui retournent S est borné par k. De manière équivalente, si l'on considère un objet qui implémente ce problème, le nombre d'opérations qui semblent s'exécuter instantanément au même instant est toujours borné par k.

Définition 2.6 ((n,k)-participating-set) *Le problème consiste, pour chaque processus p_i à obtenir un sous-ensemble S_i des identités des processus participants. Les ensembles S_i obtenus en sortie satisfont les propriétés suivantes :*

Auto inclusion *:* $\forall i : id_i \in S_i$ *;*
Comparaison *:* $\forall i,j : S_i \subseteq S_j \vee S_j \subseteq S_i$ *;*
Immédiateté *:* $\forall i,j : (id_i \in S_j) \Rightarrow (S_i \subseteq S_j)$ *;*
Simultanéité bornée *:* $\forall \ell \in [1,\ldots,n] : |\{j : |S_j| = \ell\}| \leq k$.

L'ensemble S_i obtenu par un processus p_i est, de son point de vue, l'ensemble des identités des processus qui participent ou ont participé au cours de l'exécution. Un processus se voit toujours lui même (auto inclusion). De plus, les vues S_i retournées sont totalement ordonnées par inclusion (comparaison). La propriété d'immédiateté assure que si p_i voit l'identité de p_j dans S_i, alors la vue S_j retournée par p_j est incluse dans S_i. Ainsi, si $id_i \in S_j$ et $id_j \in S_i$ alors $S_i = S_j$. Enfin, au plus k processus voient le même ensemble d'identités. Dans le problème initial, qui possède une solution sans attente dans le modèle à registres [22], la propriété simultanéité est triviale ($k = n$). À l'inverse, pour $1 \leq k < n$, ce problème n'a pas de solution dans ce contexte.

2.2 Modèle et réductions

Modèle Dans ce bref paragraphe, nous rappelons le cadre de travail et précisons les notations. Nous étudions la difficulté relative de ces problèmes dans le modèle $\mathcal{SM}_{n,n-1}[\emptyset]$: les n processus qui composent le système communiquent via des registres atomiques ; Le modèle est complètement asynchrone et dans toute exécution, au plus $n-1$ processus sont défaillants. Les processus ont des identités distinctes, mais un processus ne connaît pas initialement les identités des autres processus participants. Dans une exécution donnée, un processus est dit participant dès qu'il effectue une action visible par les autres processus (c'est-à-dire dès qu'il accède à un objet partagé). Le nombre de participants, noté p est au plus n. Le processus p_i a pour index i et pour identité id_i. L'index i n'est pas connu de p_i. Cet index est uniquement utilisé pour faciliter la présentation des algorithmes et la démonstration de leur correction.

On s'intéresse aux questions du type suivant : étant donné un problème P, existe-t-il une solution à ce problème connaissant une solution « boîte noire » au problème P' ? Pour formaliser cette notion de boîte noire, une solution à un problème P est exprimée sous forme d'objet (cf. paragraphe 1.4.4) à usage unique dont la spécification se déduit

des problèmes qu'ils implémentent. Les conventions de nommage des objets ainsi que les opérations qu'ils supportent sont résumées dans le tableau 2.3, où $\mathcal{V}$ est un ensemble arbitraire.

problème	objet	invocation	remarques
(n,k)-accord	$SA_{n,k}$	propose(v)	$v \in \mathcal{V}$
(n,k)-comité	$CD_{n,k}$	propose(v)	$v \in \mathcal{V}^k$
(n,k)-comité-binaire	$BCD_{n,k}$	propose(v)	$v \in \{0,1\}^k$
(n,k)-test&set	$TAS_{n,k}$	compete()	pas de paramètres
(n,h)-renaming	$RNA_{n,h}$	rename(id)	id est le nom originel de l'appelant
(n,k)-participating-set	$PS_{n,k}$	participate(id)	id est le nom originel de l'appelant

FIG. 2.3 – Objets étudiés

Les processus bénéficient de la solution au problème P en invoquant l'objet correspondant par l'intermédiaire d'une primitive prim(). Les valeurs retournées par ces appels suivent la spécification du problème (à condition que le nombre d'appels ne dépasse pas la capacité de l'objet et que les paramètres des appels respectent sa spécification, autrement la valeur retournée est non spécifiée). Rappelons que la notation $\mathcal{SM}_{n,t}[O]$ dénote le modèle $\mathcal{SM}_{n,n-1}[\emptyset]$ augmenté d'une infinité d'objets O. Rappelons également que la notion de réduction lire/écrire sans attente (paragraphe 1.4.4) n'impose aucune restriction sur le nombre de registres ou d'objets O.

Organisation Les réductions présentées dans les paragraphes qui suivent sont résumées dans la Figure 2.4. Étant donné deux problèmes $P1$ et $P2$, la notation $P1 \rightarrow P2$ signifie que $P2$ se réduit à $P1$.

Le paragraphe 2.3 établit l'équivalence entre les problèmes (n, f_k)-renaming et $(k+1,k)$-test&set. Au lieu de présenter un seul algorithme complexe, la construction d'un objet RNA_{n,f_k} à partir d'objets $TAS_{k+1,k}$ s'obtient en empilant une série de blocs algorithmiques élémentaires. Nous commençons par montrer (paragraphe 2.3.1) que pour le problème du test&set ensembliste, le paramètre n n'est pas pertinent : il est au moins aussi difficile d'implémenter un objet test&set ensembliste accessible par $k+1$ processus que d'implémenter cet objet pour $n > k+1$ processus. Les objets $TAS_{n,k}$ ainsi obtenus sont ensuite utilisé pour implémenter un objet $PS_{n,k}$ (paragraphe 2.3.2). L'algorithme est une simple adaptation de « l'échelle » de Borowsky et Gafni décrit dans le paragraphe 1.3.3. Ensuite, nous donnons une solution au problème du (n, f_k)-renaming fondée sur un objet $PS_{n,k}$ (paragraphe 2.3.3). Enfin, pour fermer la boucle, un algorithme élémentaire qui résout le problème du $(k+1,k)$-test&set à l'aide d'un objet RNA_{k+1,f_k} est présenté dans le paragraphe 2.3.4.

Le paragraphe 2.4 démontre principalement que les problèmes (n,k)-accord et (n,k)-comité-binaire sont équivalents. Les réductions sont encore une fois fondées sur l'utilisation d'objets (n,n)-participating-set.

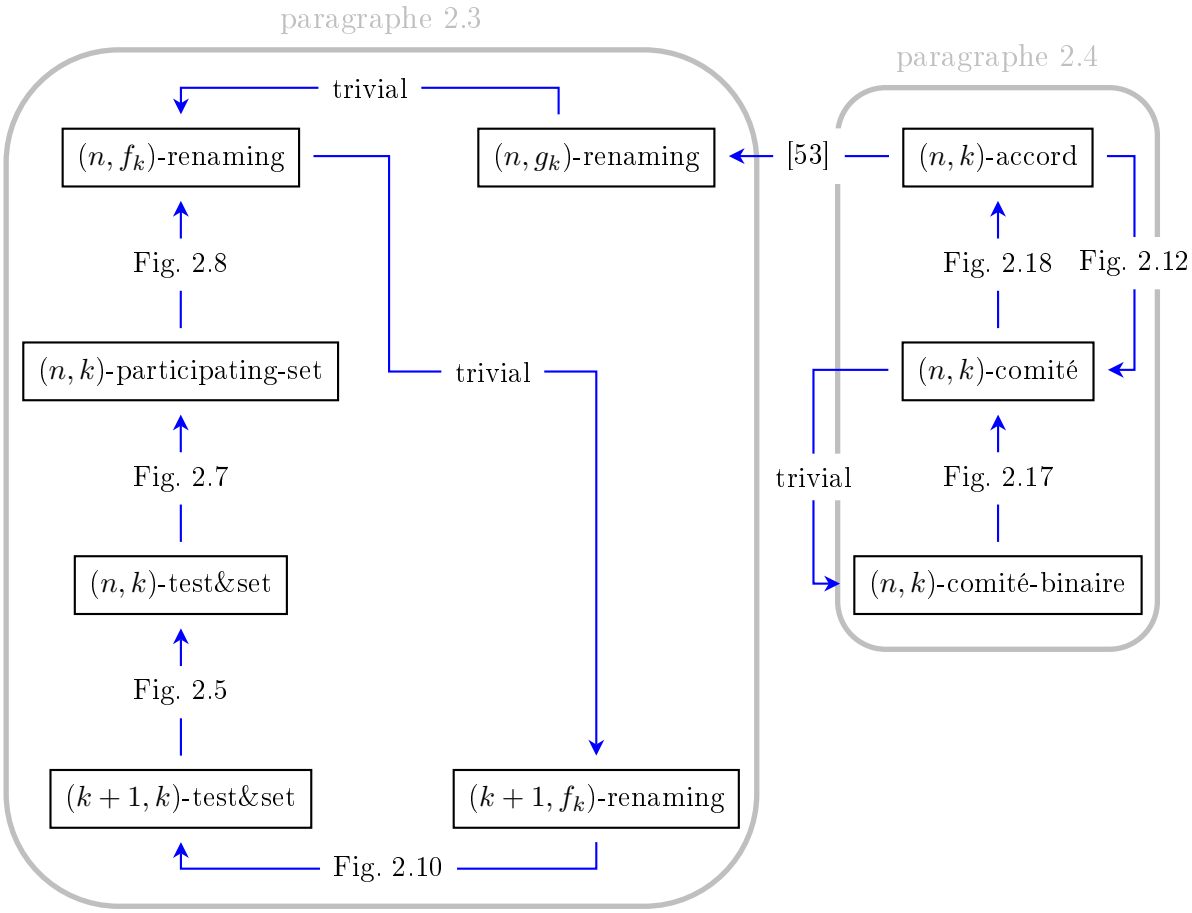

FIG. 2.4 – r/w-réductions sans attente

2.3 Accord et renommage

2.3.1 $(k+1,k)$-test&set et (n,k)-test&set

L'étude du consensus montre que n est un paramètre clé. Le rang d'un objet dans la hiérarchie [69] dépend de sa capacité à résoudre le consensus parmi un nombre donné n de processus. Au contraire, ce paragraphe établit que, lorsqu'on s'intéresse au (n,k)-test&set, le paramètre n n'est pas pertinent. Il est au moins aussi difficile de résoudre le problème (n,k)-test&set que de résoudre $(k+1,k)$-test&set. Autrement dit, dans un système équipé d'un nombre suffisant d'objets $TAS_{k+1,k}$, il existe un algorithme sans attente qui implémente un objet $TAS_{n,k}$. L'algorithme présenté ici est essentiellement dû à E. Gafni [59].

Principes de la construction L'algorithme organise un « tournoi » entre les processus. Dans chaque « match », $k+1$ processus s'affrontent en accédant à un objet $TAS_{k+1,k}$. Les vainqueurs poursuivent la compétition tandis que les perdants sortent de la compétition. Ces processus retournent alors 0 : ce sont les perdants du point de vue de l'objet $TAS_{n,k}$ implémenté. La compétition est gagnée par un processus s'il est vainqueur dans tous les matchs possibles à $k+1$ participants.

Plus précisément, les processus commencent par acquérir un nouveau nom en invoquant RNA_{n,g_n} (un tel objet peut être implémenté sans attente à partir de registres atomiques). Cette première étape est nécessaire pour organiser la compétition car les objets $TAS_{k+1,k}$ de base ne sont pas câblés statiquement et, de plus, les identités initiales des processus ne sont pas connues. Cette première étape permet aux processus de déterminer les objets de base auxquels ils devront accéder.

À l'issue de la première étape, chaque processus participant p_i dispose d'un nouveau nom new_name_i dans l'intervalle $[1,\ldots,2n-1]$. La construction utilise $\binom{2n-1}{k+1}$ objets $TAS_{k+1,k}$. Soit $\mathcal{E}$ une suite ordonnée des sous-ensembles de taille $k+1$ de $[1,2n-1]$ (l'ensemble des nouveaux noms possibles). Les objets de base $TAS_{k+1,k}$ sont indexés par les ensembles $E \in \mathcal{E}$. La suite $\mathcal{E}$ est initialement connue par chaque processus. Pour le processus p_i, la compétition se déroule comme suit : il parcourt dans l'ordre imposé par la suite $\mathcal{E}$ les ensembles E. Lorsque son nouveau nom new_name_i apparaît dans l'ensemble E courant, il accède à l'objet de base $TAS_{k+1,k}[E]$ associé. S'il obtient 0, il perd la compétition et retourne 0 (ligne 4). Autrement, il continue son parcours. Si p_i est gagnant dans tous ses « matchs » (c'est-à-dire qu'il obtient toujours 1 en réponse à ses invocations $TAS_{k+1,k}[E].\mathsf{compete}(), \forall E \in \mathcal{E}$ tel que $new\_name_i \in E$), il retourne 1 comme résultat de l'opération $TAS_{n,k}.\mathsf{compete}()$ (ligne 6).

La correction de l'algorithme est due au fait que l'ensemble des processus qui retournent 1 (à la ligne 6) se sont rencontrés dans un « match », i.e., ont tous invoqué $\mathsf{compete}()$ sur un même objet $TAS_{k+1,k}$. Ceci garantit que leur nombre est au plus k. Par exemple, la Figure 2.6 décrit une exécution possible dans le cas $k=1$ et $n=3$. Les nouveaux noms new_name_i obtenus par p_1, p_2 et p_3 à l'issue de la première étape sont respectivement 1, 2 et 4. p_2 et p_1 invoquent $\mathsf{compete}()$ sur un même objet $TAS_{2,1}$.

```
operation TAS_{n,k} .compete()
(1) new_name_i ← RNA_{n,g_n}.rename(id_i);
(2) foreach E ∈ E : new_name_i ∈ E do
(3)       ans_i ← TAS_{k+1,k}[E].compete();
(4)       if (ans_i = 0) then return(0) endif
(5) enddo
(6) return(1)
```

FIG. 2.5 – Implémentation d'un objet $TAS_{n,k}$ dans $\mathcal{SM}_{n,n-1}[TAS_{k+1,k}]$ (code pour p_i)

Le gagnant est p_2 qui continue alors son parcours. Par contre, puisque p_1 est perdant, il est éliminé de la compétition et il retourne donc 0. Dans un autre objet, p_3 (dont le nouveau nom est 4) et p_2 se rencontrent à leur tour. Le gagnant est p_3 et tous les objets sur lesquels il invoque compete() par la suite ne sont accédé que par lui. Toutes ces opérations retournent donc 1 et p_3 retourne finalement 1 comme résultat de l'opération $TAS_{3,1}$ .compete().

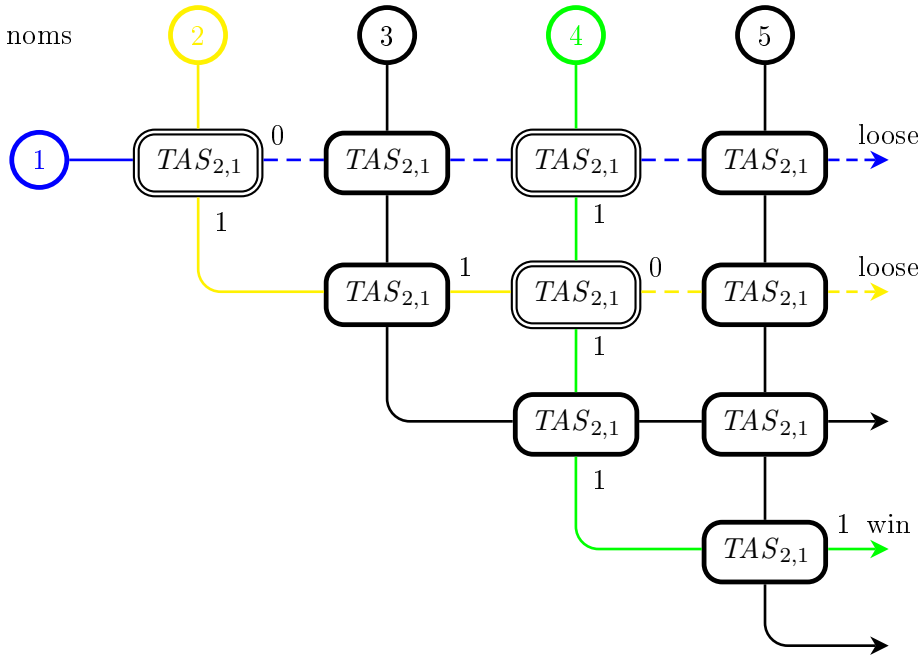

FIG. 2.6 – Construction d'un objet $TAS_{3,1}$ à partir d'objets $TAS_{2,1}$

Proposition 2.1 *L'algorithme de la Figure 2.5 est une implémentation sans attente d'un objet (n,k)-test&set dans le modèle à mémoire partagée équipé d'objets $(k+1,k)$-test&set.*

Démonstration Les propriétés validité et terminaison découlent directement du texte de l'algorithme.

- *Validité.* Lorsqu'un processus termine normalement l'algorithme (à la ligne 4 ou 6), il retourne 0 ou 1.
- *Terminaison.* Considérons un processus correct. Il obtient un nouveau nom x dans l'espace $[1, 2n-1]$ (l'espace de renommage maximal lorsque les n processus participent). Il existe $y = \binom{2n-2}{k}$ ensembles $E \in \mathcal{E}$ qui contiennent x. Au cours de la boucle **foreach**, ce processus correct accède donc au maximum à y objets $(k+1,k)$-test&set. S'il ne s'arrête pas au cours de cette boucle (en retournant 0), il retourne 1 en exécutant la dernière instruction de l'algorithme. On en déduit que l'algorithme termine et est sans attente. Dans toute exécution, le nombre d'opérations sur les objets partagés est borné par $y + 1 =$ nombre d'accès aux objets $(k+1,k)$-test&set + un accès à l'objet (n, g_n)-renaming.
- *Accord.* La démonstration de cette propriété est décomposée en deux étapes.
  - Au plus k processus sont gagnants.
    Supposons par contradiction qu'il existe une exécution dans laquelle le cardinal de l'ensemble G des gagnants est supérieur à k. Dans cette exécution, il existe donc un ensemble $E \in \mathcal{E}$ qui contient uniquement des nouveaux noms new_name_i de processus gagnants. D'après le texte de l'algorithme, chacun de ces processus invoque compete() sur l'objet $(k+1,k)$-test&set associé à E. Or, d'après la propriété d'accord de cet objet, au moins un processus obtient 0 et retourne alors 0 (à la ligne 4) : contradiction.
  - Au moins un processus est gagnant.
    Considérons une exécution arbitraire. Dans la boucle **foreach**, le processus p_i parcourt les ensembles E dans l'ordre de la suite $\mathcal{E}$. Dans l'exécution considérée, on note E_ℓ l'ensemble de plus grand rang dans la suite $\mathcal{E}$ tel qu'il existe au moins un processus qui accède à l'objet $TAS_{k+1,k}$ associé. Parmi les invocations $TAS_{k+1,k}[E_\ell]$.compete(), au moins l'une d'entre elles retourne 1 (propriété d'accord d'un objet $(k+1,k)$-test&set). Soit p_i un processus dont l'invocation $TAS_{k+1,k}[E_\ell]$.compete() retourne 1. Par définition de E_ℓ, cette invocation est la dernière effectuée par p_i dans l'exécution considérée. On en conclut, d'après le texte de l'algorithme que p_i n'est pas un perdant.

□*Proposition* 2.1

2.3.2 Objet (n,k)-participating-set

Le modèle est maintenant muni d'objets $TAS_{n,k}$. Ce paragraphe montre que le problème du (n,k)-participating-set se réduit au problème du (n,k)-test&set. Nous soulignons également certaines propriétés des objets $PS_{n,k}$ sur lesquelles sera fondé l'algorithme de renommage décrit dans le paragraphe suivant.

De l'échelle de Borowsky-Gafni à un objet (n,k)-participating-set L'algorithme (décrit dans la Figure 2.7) est fondé sur l'échelle de Borowsky-Gafni [22] que nous avons présentée dans le paragraphe 1.3.3. Il combine cette technique avec l'utilisation d'objets (n,k)-test&set pour garantir qu'au plus k processus s'arrêtent sur chaque marche. La construction est inspirée d'un algorithme donné dans [57] qui établit une

réduction entre le problème des bancs musicaux et le (n,k)-accord dans le cas $n=3$ et $k=2$.

```
init LEVEL[1..n] ← [n+1,...,n+1];
     INIT_NAME[1..n] ← [⊥,...,⊥]

operation PS_{n,k} .participate(id_i)
(01) INIT_NAME[i] ← id_i;
(02) repeat LEVEL[i] ← LEVEL[i] − 1;
(03)      foreach j ∈ {1,...,n} do level_i[j] ← LEVEL[j] enddo;
(04)      view_i ← {j : level_i[j] ≤ LEVEL[i]};
(05)      if (LEVEL[i] > k) ∧ (|view_i| = LEVEL[i])
(06)           then let ℓ = LEVEL[i];
(07)                ans_i ← TAS_{n,k}[ℓ].compete();
(08)                ok_i ← (ans_i = 1);
(09)           else ok_i ← true
(10)      endif
(11) until (|view_i| ≥ LEVEL[i]) ∧ ok_i endrepeat;
(12) let S_i = {INIT_NAME[j] : j ∈ view_i};
(13) return(S_i)
```

FIG. 2.7 – Implémentation d'un objet $PS_{n,k}$ dans $\mathcal{SM}_{n,n-1}[TAS_{n,k}]$

Lorsqu'il invoque $PS_{n,k}$ .participate(), le processus p_i fournit son identité id_i comme paramètre d'entrée. Après avoir écrit son identité dans le tableau $INIT\_NAME$, il descend « l'escalier » comme dans l'algorithme original. Cependant, pour assurer la propriété de simultanéité bornée, le prédicat qui conditionne le passage d'une marche à la suivante est différent du prédicat initial.

À chaque marche ℓ est associé un objet $TAS_{n,k}$ qui est invoqué par les processus lorsqu'ils sont bloqués à ce niveau. Seuls les processus gagnants (qui obtiennent 1 en réponse à leur invocation) sont autorisés à s'arrêter sur ce niveau. Cette règle ne remet pas en cause les propriétés de la construction originale : si $\alpha > k$ processus sont bloqués au niveau ℓ dans l'algorithme original, alors il existe suffisamment d'espace libre dans les niveaux $< \ell$ pour accueillir les β ($\alpha - k \leq \beta \leq \alpha - 1$) processus susceptibles de continuer la descente.

Plus précisément, si dans l'algorithme original, un processus p_i s'arrête au niveau $\ell \leq k$, il peut également s'arrêter à ce niveau sans violer les propriétés de l'objet $PS_{n,k}$. En effet, comme $|view_i| = \ell \leq k$ lorsque p_i stoppe sa descente, nous savons qu'au plus $\ell \leq k$ processus sont situés au niveau ℓ (ou répartis dans les niveaux inférieurs). Donc, au plus $\ell \leq k$ processus retourneront une vue de cardinal ℓ. Dans l'algorithme, ceci se traduit par la mise à vrai de la variable booléenne ok_i (lignes 05 et 09) ce qui entraîne la sortie de p_i de la boucle **repeat** (ligne 11). p_i calcule alors S_i en fonction de la valeur de $view_i$. Comme au plus $\ell \leq k$ processus exécutent ces mêmes actions qui conduisent au calcul de la même vue S_i, la propriété de simultanéité bornée est dans ce cas garantie.

Ainsi, la principale difficulté est de garantir la propriété de simultanéité bornée lorsque le niveau ℓ auquel stopperait p_i dans la construction originale est strictement plus grand que k. Comme expliqué précédemment, dans cette situation p_i fait appel

à l'objet $TAS_{n,k}[\ell]$ associé au niveau ℓ pour déterminer s'il peut s'arrêter (lignes 06-08). La propriété d'accord de cet objet assure que, parmi les processus qui devraient s'arrêter au niveau ℓ dans l'algorithme original, au moins 1 et au plus k d'entre eux stoppent effectivement à ce niveau. Si un processus p_i n'est pas autorisé à s'arrêter (nous avons alors $ok_i = \mathit{false}$), il doit poursuivre sa descente (ligne 11). Dans l'algorithme original, lorsqu'un processus stoppe au niveau ℓ, le nombre de processus situés sur les niveaux $\ell' \leq \ell$ est exactement ℓ. Bien que certains processus soient éventuellement forcés de poursuivre leur descente, cette propriété est maintenue. Ceci provient du fait que lorsqu'un processus est contraint de passer du niveau $\ell > k$ au niveau $\ell - 1$, il existe au moins un processus qui stoppe au niveau ℓ car l'objet $TAS_{n,k}[\ell]$ désigne toujours au moins un gagnant.

Notations et propriétés Afin de préparer le terrain pour la construction suivante, nous introduisons des notations ainsi qu'une propriété qui caractérise la répartition des processus sur les marches. Cette propriété découle directement de la spécification d'un des objets (n,k)-participating-set.

Soit S_i la vue obtenue par p_i après avoir invoqué participate(id_i) et $\ell = |S_i|$. On appelle ℓ le *niveau* de p_i, et on dira que « p_i est bloqué ou est au niveau ℓ ». S'il existe un processus p_j tel que $|S_j| = \ell$, on écrira que « le niveau ℓ n'est pas vide ». Dans le cas contraire, on écrira que le niveau ℓ est vide. On note $\mathcal{L}$ l'ensemble des niveaux non vides, $|\mathcal{L}| = m \leq n$. Nous ordonnons ces m niveaux par ordre croissant, i.e., $\mathcal{L} = \{\ell_1 < \ell_2 < \ldots < \ell_m\}$. Les niveaux $\ell \in \{1, \ldots, n\} - \mathcal{L}$ sont vides. Ainsi, dans l'exemple donné dans la table 2.1, $\mathcal{L} = \{10, 8, 5, 3, 2\}$ et on a donc $\ell_1 = 2, \ell_2 = 3, \ell_3 = 5, \ell_4 = 8$ et $\ell_5 = 10$.

Lorsque p_j est bloqué au niveau ℓ (i.e., $|S_j| = \ell$), il y a de son point de vue exactement ℓ processus dans l'exécution courante. L'ensemble des identités de ces processus constitue la vue S_j de p_j. Ces processus se répartissent sur les niveaux $\ell', 1 \leq \ell' \leq \ell$. En effet, la propriété d'immédiateté assure que si $id_x \in S_j$ alors $S_x \subseteq S_j$. Donc, le niveau de p_x qui est égal à $|S_x|$ est inférieur ou égal à $\ell = |S_j|$. De même, les propriétés suivantes découlent facilement de la définition du (n,k)-participating-set :

- Soit p le nombre de processus qui invoquent participate(). Le niveau d'un processus qui obtient une réponse est inférieur ou égal à p.
- Les processus bloqués au même niveau ℓ partagent la même vue (propriété de comparaison).
- Soient p_i et p_j tels que $|S_i| = \ell_x$ et $|S_j| = \ell_y$. Supposons que $\ell_x < \ell_y$. On a alors :
  - $S_i \subset S_j$ (car $|S_i| = \ell_x < \ell_y = |S_j|$ et les deux vues sont comparables par inclusion) ;
  - $|S_j - S_i| = |S_j| - |S_i| = \ell_y - \ell_x$ (conséquence de la remarque précédente).

Un objet (n,k)-participating-set « répartit » les processus sur des niveaux allant de 1 à p, où p est le nombre de processus qui accèdent à l'objet. Cette répartition est telle que (1) le nombre de processus par niveau est au plus k et (2) les processus obtiennent des vues « cohérentes » de la répartition. Par exemple, la Table 2.1 donne une répartition possible, résultat de l'invocation d'un objet $PS_{n,3}$ par 10 processus. Dans cette exécution, aucun processus ne tombe en panne. Pour simplifier, on suppose que l'iden-

tité d'un processus est son index (chaque processus p_i invoque $PS_{n,3}.\mathsf{participate}(i)$). Par exemple, les deux processus p_2 et p_8 sont bloqués au niveau 5. Leur vue est l'ensemble des processus bloqués au niveau 5 ou aux niveaux inférieurs. Certains niveaux sont vides.

niveau	processus	vues S_i
10	p_5, p_9	$S_5 = S_9 = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}\}$
9		niveau vide
8	p_1, p_3, p_10	$S_1 = S_3 = S_{10} = \{p_1, p_2, p_3, p_4, p_6, p_7, p_8, p_{10}\}$
7		niveau vide
6		niveau vide
5	p_2, p_8	$S_2 = S_8 = \{p_2, p_4, p_6, p_7, p_8\}$
4		niveau vide
3	p_7	$S_7 = \{p_4, p_6, p_7\}$
2	p_4, p_6	$S_4 = S_6 = \{p_4, p_6\}$
1		niveau vide

TAB. 2.1 – Objet $PS_{n,k}$: exemple de vues retournées ($|P| = 10 \leq n, k = 3$)

Les propriétés de l'objet n'autorisent pas n'importe quelle répartition arbitraire des processus sur les niveaux. Par exemple, modifions la Table 2.1 en plaçant le processus p_8 au niveau 6. Il est clair que cette répartition n'est pas valide puisque qu'on aurait alors $|S_8| = |\{p_2, p_4, p_6, p_7, p_8\}| = 5$ ce qui contredit la position de p_8 dans l'échelle des niveaux. Intuitivement, la présence de trois processus au niveau 8 empêche l'occupation des deux niveaux inférieurs 7 et 6. Le lemme suivant formalise cette idée. Nous notons $B[\ell_x] = \{j : |S_j| = \ell_x\}$ l'ensemble des processus bloqués au niveau ℓ_x. Pour simplifier, on définit également le niveau fictif $\ell_0 = 0$. On a toujours $B[\ell_0] = \emptyset$.

Lemme 2.1 $\forall \ell_x \in \mathcal{L} : |B[\ell_x]| \leq \min(k, \ell_x - \ell_{x-1})$.

Démonstration La propriété de simultanéité bornée se traduit par $|B[\ell_x]| \leq k$. Soient p_i et p_j deux processus bloqués aux niveaux ℓ_x et ℓ_{x-1} respectivement. Si $\ell_{x-1} = \ell_0$, l'inégalité se ramène à l'observation qu'au plus ℓ processus sont au niveau ℓ. Nous avons :

1. $|S_i| = \ell_x$ et $|S_j| = \ell_{x-1}$ (définition d'un processus bloqué à un niveau) ;
2. $B[\ell_x] \subseteq S_i$ et $B[\ell_{x-1}] \subseteq S_j$ (propriétés d'auto-inclusion et comparaison) ;
3. $B[\ell_x] \cap B[\ell_{x-1}] = \emptyset$ (définition d'un processus bloqué à un niveau) ;
4. $B[\ell_x] \cap S_j = \emptyset$ et $B[\ell_x] \cup B[\ell_{x-1}] \subseteq S_i$ (point 3 et $S_i \subsetneq S_j$) ;
5. $B[\ell_x] = (B[\ell_x] \cup B[\ell_{x-1}]) - B[\ell_{x-1}] \subseteq S_i - S_j$ (point 4) ;
6. Nous en concluons : $|B[\ell_x]| \leq |S_i - S_j| \leq \ell_x - \ell_{x-1}$.

□*Lemme* 2.1

2.3.3 Un algorithme de (n, f_k)-renaming adaptatif

Un algorithme qui implémente un objet (n, f_k)-renaming est décrit dans la Figure 2.8. Pour obtenir un nouveau nom, un processus p_i appelle rename(id_i) où id_i est son nom initial. Dans une exécution où $p \leq n$ processus invoquent rename(id), p_i obtient un nouveau nom dans l'intervalle $[0, \ldots, 2p - \lceil \frac{p}{k} \rceil] = [0, \ldots, f_k(p)]$ lorsqu'il exécute la ligne 5.

Objets de base La construction est fondée sur un objet (n, k)-participating-set et n objets (k, g_k)-renaming notés $RNA_{k,g_k}[1], \ldots, RNA_{k,g_k}[n]$. Rappelons la définition de la fonction g_k : $g_k : p \rightarrow \min(2p-1, p+k-1)$. Ainsi, $\forall p \leq k : g_k(p) = 2p-1$. Il s'en suit que les objets RNA_{k,g_k} peuvent être implémentés sans attente en utilisant uniquement des registres atomiques (cf. Figure 2.2 et paragraphe 2.1.4).

La spécification d'un objet (k, g_k)-renaming requiert qu'il soit accédé par au plus k processus. Chaque objet $RNA_{k,g_k}[x]$.rename(id) attribue des nouveaux noms dans l'intervalle $[0, g_k(p)]$, où p est le nombre de processus qui accèdent à l'objet d'index x. La construction repose sur les deux idées simples suivantes :

- Bénéficier de la répartition des processus sur les marches assurée par l'objet $PS_{n,k}$ pour garantir qu'au plus k processus accéderont à chaque objet RNA_{k,g_k} ;
- Translater les noms obtenus via les objets RNA_{k,g_k} pour éviter les conflits dans l'attribution du nouveau nom final.

L'algorithme : principes et description La construction s'inspire de l'algorithme de renommage de Borowsky et Gafni [22]. Il applique la stratégie classique « diviser pour régner ».

```
operation RNA_{n,f_k}.rename(id_i)
(1) S_i ← PS_{n,k}.participate(id_i);
(2) base_i ← (2 × |S_i| − ⌈|S_i|/k⌉);
(3) offset_i ← RNA_{k,g_k}[|S_i|].rename(id_i);
(4) new_name_i ← base_i − offset_i + 1;
(5) return(new_name_i)
```

FIG. 2.8 – (n, f_k)-renaming dans $\mathcal{SM}_{n,n-1}[PS_{n,k}]$ (code pour p_i)

- Diviser pour régner.
  Un processus p_i commence par invoquer l'objet (n, k)-participating-set pour obtenir une vue S_i (ligne 1). L'objectif est de répartir les processus dans des ensembles deux à deux disjoints. Les partitions sont définies à partir des vues S_i obtenues : les processus qui obtiennent la même vue S sont dans la même partition. Les vues S_i satisfont les propriétés d'auto inclusion, de comparaison, d'immédiateté et de simultanéité bornée. Nous déduisons de ces propriétés que (1) le nombre de partitions est majoré par le nombre de processus participants p et (2) qu'il existe au plus k processus par partition.

Tous les processus qui obtiennent la même vue S sont bloqués au même niveau ℓ dans l'objet $PS_{n,k}$. Une façon simple d'identifier la partition à laquelle p_i appartient est de considérer le niveau ℓ atteint par p_i dans l'objet $PS_{n,k}$ ($\ell = |S_i|$). Les $\alpha \leq k$ processus dans la partition $\ell = |S|$ entrent alors en compétition pour acquérir un nouveau nom (voir la Figure 2.9). Pour ce faire, chaque processus p_i accède à l'objet (k, g_k)-renaming correspondant à sa partition $\ell = |S_i|$ en invoquant $RNA_{k,g_k}[|S_i|].\mathsf{rename}(id_i)$ (ligne 3). Comme nous l'avons remarqué précédemment, les nouveaux noms sont attribués dans l'intervalle $[0, 2\alpha - 1]$.

– Recoller les morceaux.
L'attribution du nom final est similaire à un mode d'adressage basé. À chaque partition $\ell = |S_i|$ est associée la base $2 \times |S_i| - \lceil \frac{|S_i|}{k} \rceil$ (ligne 2). Les bases associées à deux partitions différentes ne sont donc pas égales. Ensuite, l'entier résultat de l'invocation $RNA_{k,g_k}[\ell].\mathsf{rename}(id_i)$ est vu comme un déplacement (offset_i). p_i détermine son nouveau nom final à l'aide de la base base_i et du déplacement offset_i en considérant l'espace de renommage descendant qui démarre à l'adresse base_i (ligne 4, voir aussi la Figure 2.9).
Remarquons qu'au sein d'une partition, les processus calculent la même base alors que les déplacements offset_i sont deux à deux distincts (propriété d'unicité garantie par les objets de base RNA_{k,g_k}). Nous en déduisons que les noms $\mathit{new\_name}_i$ calculés par ces processus sont deux à deux distincts. Il reste cependant à vérifier que l'attribution de ces noms n'entre pas en conflit avec les noms calculés dans les autres partitions. Le Lemme 2.2 établit que les espaces de renommage utilisés dans chaque partition sont deux à deux disjoints.

Lemme 2.2 *L'algorithme de la Figure 2.8 garantit que les nouveaux noms attribués sont uniques : quelque soit l'exécution, il n'existe pas deux processus qui obtiennent le même nom.*

Démonstration Pour commencer, vérifions que les objets de base RNA_{k,g_k} sont utilisés conformément à leur spécification, c'est-à-dire accédés par au plus k processus. D'après le texte de l'algorithme, seuls les processus p_i tels que $|S_i| = \ell_x$ appellent l'objet $RNA_{k,g_k}[\ell_x]$. L'objet $PS_{n,k}$ assure que ces processus sont au plus k (propriété de simultanéité bornée).

Soit p_i un processus tel que $|S_i| = \ell_x$. Ce processus appartient à l'ensemble $B[\ell_x]$ des processus bloqués au niveau ℓ_x. Pour obtenir le déplacement offset_i, il invoque $\mathsf{rename}(id_i)$ sur l'objet (k, g_k)-renaming d'index ℓ_x. D'après les propriétés de cet objet, il n'existe pas deux processus dans $B[\ell_x]$ qui obtiennent le même déplacement et on a $1 \leq \mathit{offset}_i \leq g_k(|B[\ell_x]|) = 2 \times |B[\ell_x]| - 1$ car $1 \leq B[\ell_x] \leq k$. Par ailleurs, on sait que $|B[\ell_x]| \leq \min(k, \ell_x - \ell_{x-1})$ (Lemme 2.1). L'inégalité devient donc $1 \leq \mathit{offset}_i \leq 2 \times \min(k, \ell_x - \ell_{x-1})$.

D'un autre coté, l'espace de renommage attribué au processus $p_i \in B[\ell_x]$ débute à la base $2\ell_x - \lceil \frac{\ell_x}{k} \rceil$ (inclus) et descend jusqu'à $2\ell_{x-1} - \lceil \frac{\ell_{x-1}}{k} \rceil$ (exclus). La taille de cet espace est donc :

$$2(\ell_x - \ell_{x-1}) - (\lceil \frac{\ell_x}{k} \rceil - \lceil \frac{\ell_{x-1}}{k} \rceil)$$

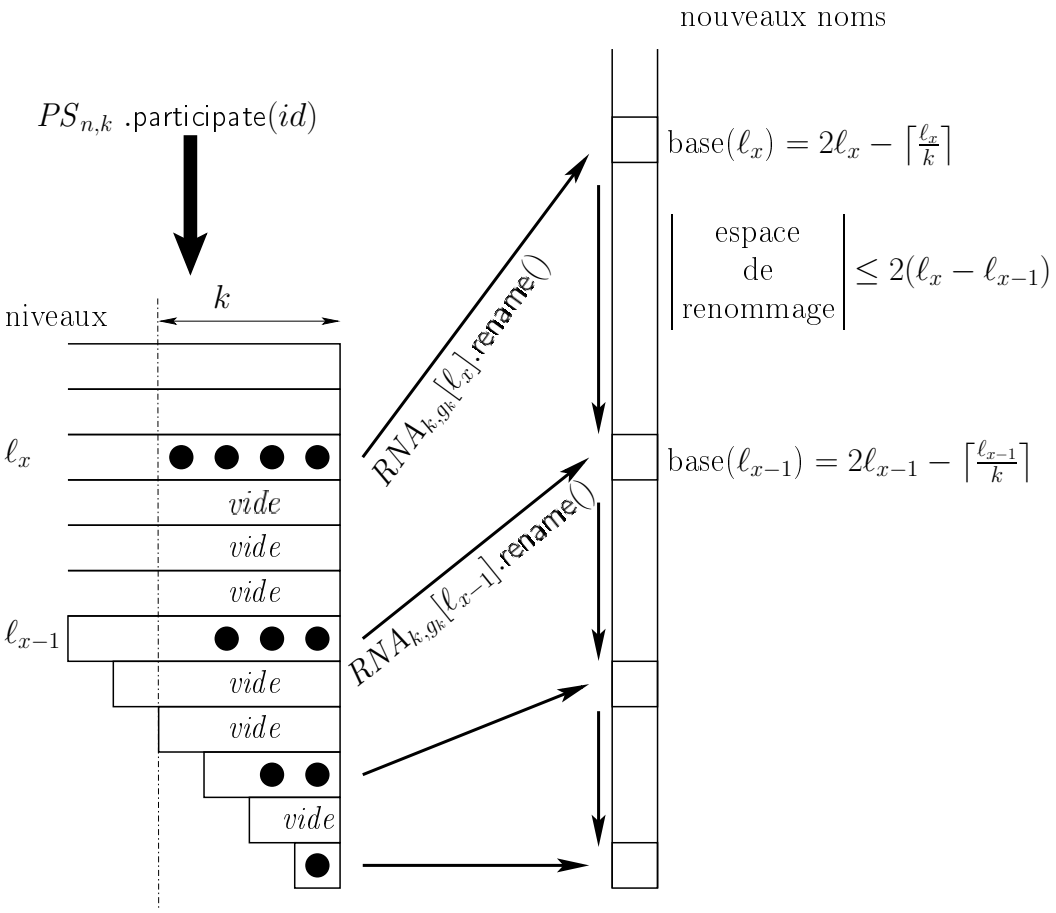

FIG. 2.9 – Principe de l'algorithme de renommage adaptatif

On en déduit qu'une condition suffisante pour éviter les conflits dans l'attribution des nouveaux noms est que le déplacement maximal soit inférieur ou égal à la taille de l'espace de renommage, soit :

$$2 \times \min(k, \ell_x - \ell_{x-1}) - 1 \leq 2(\ell_x - \ell_{x-1}) - (\lceil \frac{\ell_x}{k} \rceil - \lceil \frac{\ell_{x-1}}{k} \rceil) \qquad (2.1)$$

Soient q_x, r_x (respectivement q_{x-1}, r_{x-1}) le quotient et le reste dans la division euclidienne de ℓ_x (respectivement ℓ_{x-1}) par k. On écrit :

$$\begin{aligned}
\ell_x &= q_x k + r_x \quad (\text{d'où } \lceil \tfrac{r_x}{k} \rceil \in \{0,1\}) \\
\ell_{x-1} &= q_{x-1} k + r_{x-1} \quad (\text{d'où } \lceil \tfrac{r_{x-1}}{k} \rceil \in \{0,1\}) \\
\text{d'où} \quad \ell_x - \ell_{x-1} &= (q_x - q_{x-1})k + (r_x - r_{x-1}) \quad \text{et} \\
(\lceil \tfrac{\ell_x}{k} \rceil - \lceil \tfrac{\ell_{x-1}}{k} \rceil) &= (q_x - q_{x-1}) + (\lceil \tfrac{r_x}{k} \rceil - \lceil \tfrac{r_{x-1}}{k} \rceil)
\end{aligned}$$

Remarquons que $-(k-1) \leq r_x - r_{x-1} \leq k-1$ et que $q_x \geq q_{x-1}$ car $\ell_x > \ell_{x-1}$. Nous distinguons deux cas en fonction de la valeur de la différence entre q_x et q_{x-1}.

- $q_x - q_{x-1} = 0$
  Dans ce cas $-(k-1) \leq \ell_x - \ell_{x-1} = r_x - r_{x-1} \leq k-1$ et l'inégalité 2.1 devient

$$\begin{aligned} -1 &\leq -(q_x - q_{x-1}) - (\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil) \text{ c'est-à-dire} \\ \lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil &\leq 1 \end{aligned}$$

  Cette inégalité est vérifiée car $\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil \in \{-1, 0, 1\}$.
- $q_x - q_{x-1} = 1$
  On distingue deux cas en fonction de la valeur de la différence $r_x - r_{x-1}$.
  - $r_x - r_{x-1} \leq 0$
    L'équation 2.1 devient :

$$\begin{aligned} -1 &\leq -(q_x - q_{x-1}) - (\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil) \text{ qui donne} \\ (\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil) &\leq 0 \end{aligned}$$

    $(\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil) \in \{-1, 0\}$ car $r_x \leq r_{x-1}$. L'inégalité est donc vérifiée.
  - $r_x - r_{x-1} > 0$
    Dans ce cas, $\min(k, \ell_x - \ell_{x-1}) = k$. L'équation 2.1 devient :

$$\begin{aligned} 2k-1 &\leq 2\big((q_x - q_{x-1})k + (r_x - r_{x-1})\big) - (q_x - q_{x-1}) - (\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil) \\ 0 &\leq 2(r_x - r_{x-1}) - (\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil) \\ (\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil) &\leq 2(r_x - r_{x-1}) \end{aligned}$$

    On a toujours $(\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil) \in \{0, 1\}$. D'autre part, $2 \leq 2(r_x - r_{x-1})$ car $r_x - r_{x-1} > 0$. L'inégalité est bien vérifiée.
- $q_x - q_{x-1} = \alpha > 1$
  Dans ce cas, $\min(k, \ell_x - \ell_{x-1}) = k$ et la partie gauche (1) de l'inégalité 2.1 devient $2k-1$. La partie droite (2) est égale à :

$$\begin{aligned} (2) &= 2\big(k(q_x - q_{x-1}) + (r_x - r_{x-1})\big) - \big((q_x - q_{x-1}) + (\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil)\big) \\ &= 2\big(k\alpha + (r_x - r_{x-1})\big) - (\alpha + (\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil)) \\ &= (2k-1)\alpha + 2(r_x - r_{x-1}) - (\lceil \frac{r_x}{k} \rceil - \lceil \frac{r_{x-1}}{k} \rceil) \text{ d'où} \\ (2) &\geq (2k-1)\alpha - 2(k-1) - 1 = (2k-1)(\alpha - 1) \end{aligned}$$

  Comme $\alpha > 1$, on obtient $(1) \leq (2)$: l'équation 2.1 est vérifiée.

□*Lemme* 2.2

Proposition 2.2 *L'algorithme décrit dans la Figure 2.8 est une implémentation sans attente d'un objet RNA_{n,f_k} dans le modèle à mémoire partagée équipé d'objets (n,k)-participating-set et (k,g_k)-renaming.*

Démonstration La propriété sans attente de l'implémentation découle du texte de l'algorithme et du fait que les objets de base $PS_{n,k}$ et RNA_{k,g_k} le sont. Le Lemme 2.2 démontre l'unicité des noms retournés. Il reste à vérifier que les noms attribués lorsque $p \leq n$ processus participent sont dans l'intervalle $[1, f_k(p)]$.

Lorsque p processus participent, la taille des vues S_i est majorée par p. La plus grande base $base_i$ calculée est donc $2 \times p - \lceil \frac{p}{k} \rceil$, ce qui est aussi la plus grande valeur possible pour un nouveau nom (ligne 4). L'espace de renommage est donc $[1, 2 \times p - \lceil \frac{p}{k} \rceil] = [1, f_k(p)]$. $\square_{Proposition\ 2.2}$

2.3.4 Du $(k+1, f_k)$-renaming vers $(k+1, k)$-test&set

Dans les paragraphes précédents, nous avons montré comment construire un objet (n, f_k)-renaming à partir d'objets $(k+1, k)$-test&set et de registres atomiques. L'algorithme simple décrit dans la Figure 2.10 ferme la boucle. Il implémente de façon sans attente un objet $(k+1, k)$-test&set à partir d'un objet $(k+1, f_k)$-renaming. Pour décider, un processus commence par acquérir un nouveau nom en invoquant RNA_{k+1,f_k}.rename(id_i). Il retourne 1 si et seulement si son nouveau nom est compris entre 1 et k.

```
operation TAS_{k+1,k} .compete()
(1) new_name_i ← RNA_{k+1,f_k}.rename(id_i);
(2) if (new_name_i ≤ k) then return(1)
(3)                     else return(0)
(4) endif
```

FIG. 2.10 – $(k+1, k)$-test&set dans $\mathcal{SM}_{k+1,k}[RNA_{k+1,f_k}]$ (code pour p_i)

Proposition 2.3 *L'algorithme décrit dans la Figure 2.10 implémente sans attente un objet $(k+1, k)$-test&set dans le modèle à mémoire partagée équipé d'un objet $(k+1, f_k)$-renaming.*

Démonstration Les propriétés de validité et de terminaison découlent immédiatement du code. De même, il est évident que le nombre de 1 retournés est borné par k. Nous devons montrer qu'au moins un processus retourne 1. Soit $p \leq k+1$ le nombre de processus participants. Nous considérons deux cas en fonction de la valeur de p.

- $p = k+1$ processus invoquent RNA_{k+1,f_k} .rename(id_i)
  Dans ce cas, la taille M de l'espace de renommage est $M = f_k(k+1) = 2(k+1) - \lceil \frac{k+1}{k} \rceil = 2k$. Puisque les nouveaux noms sont attribués dans l'intervalle $[1, 2k]$, il existe au moins un processus parmi les $k+1$ participants qui obtient un nouveau nom $\leq k$. On en déduit qu'il existe au moins un processus gagnant.

– $p \leq k+1$ processus invoquent RNA_{k+1,f_k} .rename(id_i)
On a alors $M = f_k(p) = 2p - \lceil \frac{p}{k} \rceil = p + (p-1)$. On en déduit qu'au moins un processus obtient un nouveau nom $\in [1,p]$. Ce processus est un gagnant car $p \leq k$.

□*Proposition* 2.3

2.4 Accord et décision en comité

Dans ce paragraphe, nous établissons que l'accord sur plusieurs fronts et l'accord ensembliste sont les facettes d'un même problème. Nous montrons que, pour tout k ($1 \leq k \leq n$) il existe une réduction sans attente du problème du (n,k)-accord vers le (n,k)-comité et réciproquement. De plus, nous étendons ces résultats à la version binaire du problème de décision de comité : il existe un algorithme sans attente, fondé sur des registres et des objets (n,k)-comité-binaire qui résout le problème du (n,k)-accord ; Réciproquement, à partir de registres et d'objets (n,k)-accord, il est possible de construire un objet (n,k)-comité-binaire (cf. Figure 2.11).

Ainsi, le problème (n,k)-comité-binaire caractérise de façon fine le (n,k)-accord, de la même façon que le consensus binaire est équivalent au consensus. (n,k)-comité-binaire est donc « l'accord ensembliste binaire ». Ces résultats sont parus dans [4].

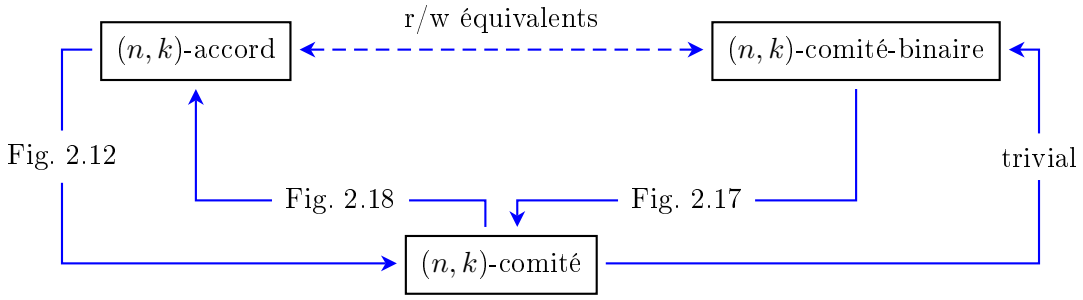

FIG. 2.11 – Réductions entre accord ensembliste et accord sur plusieurs fronts

Les constructions des objets (n,k)-comité reposent sur l'idée simple suivante : *réduire le nombre de vecteurs*. Considérons le problème du (n,k)-accord. Si l'on sait que les processus démarrent avec au plus k valeurs distinctes, alors il existe une solution triviale, et ce quelque soit le nombre de processus : chaque participant décide simplement la valeur qu'il propose. De façon similaire, nous pouvons résoudre le problème (n,k)-comité en utilisant uniquement des registres, et pour un nombre quelconque de processus, lorsque le nombre de vecteurs proposés est borné par k.

Plus précisément, cette idée se traduit par le problème $([n],k)$-BG. Les paramètres n et k font référence respectivement au nombre initial et final de valeurs. Le paramètre n est encadré pour souligner que l'on veut réduire l'espace des valeurs, indépendamment du nombre de processus participants. Dans ce problème, un nombre arbitraire de processus proposent une valeur telle que le nombre de valeurs proposées distinctes est inférieur ou égal à n. Comme dans le problème du (n,k)-comité, en sortie les processus doivent produire un couple (numéro de comité, valeur) tel que, dans chaque comité, au

plus une valeur est décidée. Les problèmes (n,k)-comité et $([n],k)$-BG sont très similaires. Cependant, la spécification du $([n],k)$-BG ne limite pas le nombre de processus participants mais impose une contrainte sur l'espace des valeurs proposées. La solution algorithme à ce problème pour le cas $n = k$ est la clé des réductions résumées dans la Figure 2.11.

Définition 2.7 ($([n],k)$-BG) *Chaque processus p_i propose en entrée une valeur v_i telle que :*

Espace d'entrée borné $|\{v_i : p_i \text{ participe }\}| \leq n$.

En sortie, pour chaque processus p_i, le problème consiste à décider un couple (c_i, d_i), $1 \leq c_i \leq k$, qui respecte les propriétés suivantes :

Validité $\exists p_j$ *processus participant tel que* $d_i = v_j$;

Terminaison *Tout processus correct qui participe décide ;*

Accord $\forall i,j : c_i = c_j \Rightarrow d_i = d_j$.

Le nom du problème fait référence à la BG-simulation [21, 25]. Cet outil algorithmique puissant permet par exemple d'étudier le calcul t-tolérant par réduction au calcul sans attente [21] ou d'obtenir des bornes de complexité dans le modèle synchrone à partir de résultats de calculabilité asynchrone [54, 50]. Sur ce dernier point, [101] donne des exemples simples d'utilisation. La spécification ci-dessus est une définition alternative à la BG-simulation.

[4] étudie également une généralisation moins contrainte du problème appelée $([n],k,\alpha)$-BG-étendu. La propriété d'accord est relâchée de la façon suivante : dans chaque comité, au plus α valeurs peuvent être décidées. Il est démontré que ce problème a une solution sans-attente pour un nombre quelconque de processus en utilisant uniquement des registres lorsque le nombre de valeurs initiales n satisfait la relation $k \geq \lceil \frac{n}{\alpha} \rceil$.

Organisation La technique centrale, sur laquelle sont fondées les réductions de cette partie est l'algorithme sans attente qui résout le $([k],k)$-BG dans le modèle $\mathcal{SM}_{n,n-1}[\emptyset]$. Cet algorithme est décrit dans le paragraphe 2.4.2, à la Figure 2.13. Le paragraphe 2.4.1 en montre une première application en présentant une construction sans attente du (n,k)-comité fondée sur l'utilisation conjointe d'un objet $([k],k)$-BG et d'un objet (n,k)-accord. Ensuite, nous montrons comment, à l'aide d'un objet (n,k)-comité-binaire, il est possible de résoudre le problème $([k+1],k)$-BG. L'algorithme 2.13 est à la base de cette nouvelle construction. Finalement, pour compléter ces réductions, l'algorithme trivial qui implémente un objet (n,k)-accord à partir du (n,k)-comité est brièvement évoqué dans le paragraphe 2.4.5.

2.4.1 Du (n,k)-accord vers le (n,k)-comité

Supposons que l'on dispose d'une solution sans attente au problème du $([k],k)$-BG pour n processus (c'est-à-dire un objet $BG_{[k],k}$ accessible par n processus). Il est clair que l'on peut alors résoudre le (n,k)-comité à partir du (n,k)-accord. Les processus démarrent en invoquant l'objet $SA_{n,k}$ pour globalement réduire l'espace des vecteurs proposés à au plus k vecteurs. Ensuite, chaque processus p_i appelle $BG_{[k],k}.\mathsf{propose}(prop_i)$,

où $prop_i$ est le vecteur résultat de l'invocation de l'objet $SA_{n,k}$ par p_i, pour obtenir un numéro de comité c_i ainsi qu'un vecteur d_i. À chaque comité c est associé au plus un vecteur d. p_i décide donc finalement $(c_i, d_i[c_i])$, conformément à la spécification du (n,k)-comité.

Cet algorithme simple est décrit dans la Figure 2.12. Un processus p_i accède à l'objet en appelant $CD_{n,k}.\mathsf{propose}(vec_i)$ où $vec_i \in \mathcal{V}^k$ est un vecteur de k valeurs. Il décide une paire (numéro de comité, valeur) lorsqu'il exécute la ligne 3. Une construction sans-attente d'un objet $BG_{[k],k}$ à partir de registres atomiques est présentée dans le paragraphe 2.4.2 (Figure 2.13).

```
operation CD_{n,k}.propose(vec_i)
(1) prop_i ← SA_{n,k}.propose(vec_i);
(2) (c_i, d_i) ← BG_{[k],k}.propose(prop_i);
(3) return((c_i, d_i[c_i]))
```

FIG. 2.12 – (n,k)-comité dans $\mathcal{SM}_{n,n-1}[BG_{[k],k}]$ (code pour p_i)

La correction de l'algorithme découle facilement du code et des propriétés des objets $SA_{n,k}$ et $BG_{[k],k}$ sous-jacents. Les propriétés de terminaison de ces objets impliquent que l'implémentation est sans attente. De même, les propriétés de validité garantissent que $d_i[c_i]$ est une valeur proposée pour le comité c_i. Pour l'accord, considérons deux processus p_i et p_j qui décident respectivement (c, v_i) et (c, v_j). On alors $v_i = d_i[c]$ et $v_j = d_j[c]$. D'après la propriété d'accord de l'objet $BG_{[k],k}$, $d_i = d_j$, d'où $v_i = v_j$.

2.4.2 Une solution sans attente au problème du $([k], k)$-BG

L'algorithme de la Figure 2.13 construit un objet $BG_{[k],k}$ à partir de registres atomiques. L'objet est accessible par au plus n processus et requiert que le nombre total de valeurs v_i proposées soit borné par k. Le processus p_i invoque l'objet en appelant $BG_{[k],k}.\mathsf{propose}(v_i)$. Il décide une paire (numéro de comité, valeur) lorsqu'il exécute la ligne 7.

Objets partagés La construction repose sur k objets (n,n)-participating-set (notés $PS_{n,n}[1], \ldots, PS_{n,n}[k]$). Les objets $PS_{n,n}$ sont construits à partir de registres atomiques. En effet, dans le problème (n,n)-participating-set, la contrainte simultanéité bornée disparaît. En l'absence de cette contrainte, il existe une solution sans attente fondée uniquement sur des registres atomiques (i.e., l'algorithme de l'échelle de Borowsky Gafni).

Principe Chaque processus démarre l'algorithme avec une valeur $v_i \in \mathcal{V}$ et doit décider une paire (c_i, d_i) composée d'un numéro de comité c_i $(1 \leq c_i \leq k)$ et d'une valeur proposée d_i. L'algorithme doit assurer que pour chaque comité, au plus une valeur est décidée (c'est-à-dire : $\forall i, j : c_i = c_j \Rightarrow d_i = d_j$). L'ensemble $\mathcal{V}$ dans lequel sont choisies les valeurs proposées est arbitraire. On supposera cependant qu'il s'agit d'un ensemble totalement ordonné.

```
operation BG_{[k],k}.propose(v_i)
(1)  prop_i ← v_i ;
(2)  for r_i = 1 to k do
(3)      S_i ← PS_{n,n}[r_i].participate(< id_i, prop_i >) ;
(4)      view_i ← {prop_j :< id_j, prop_j >∈ S_i} ;
(5)      if (|view_i| > 1) then prop_i ← min(view_i)
(6)                        else let v such that view_i = {v} ;
(7)                             return((r_i, v))
(8)      endif
(9)  enddo
```

FIG. 2.13 – $([n], k)$-BG dans $\mathcal{SM}_{n,n-1}[\emptyset]$ (code pour p_i)

L'algorithme procède par rondes asynchrones numérotées $1, \ldots, k$. Les processus qui participent à la ronde r essaient de décider dans le r-ième comité. Un processus p_i exécute successivement les rondes $1, 2, \ldots$ jusqu'à ce qu'il parvienne à décider en exécutant la ligne 7. Si, lorsque p_i décide $r_i = r$, on écrira que « p_i décide dans le comité r ». L'algorithme est correct s'il garantit que (1) les processus qui décident lors d'une même ronde r décident la même valeur et (2) tout processus qui ne défaille pas parvient à décider dans un comité.

La variable locale $prop_i$ contient en permanence une valeur initialement proposée. La sémantique de cette variable est la suivante : s'il participe à la ronde r, $prop_i$ est la valeur proposée par p_i dans le comité r. Initialement, $prop_i$ est la valeur proposée par p_i (ligne 1). Les processus démarrent donc les rondes avec collectivement au plus k valeurs distinctes.

Lors de la ronde r, le processus p_i calcule d'abord une vue $view_i$ des valeurs proposées dans le comité r à l'aide de l'objet $PS_{n,n}[r]$ (lignes 3-4) associé à la ronde r. Grâce aux propriétés de l'objet $PS_{n,n}$, les vues sont ordonnées par inclusion et toutes les vues de taille ℓ sont donc identiques. Si le cardinal de la vue de p_i est égal à 1, il décide dans le comité r l'unique valeur v contenue dans cette vue (ligne 6-7). Puisque que les vues S_i sont ordonnées par inclusion, il existe au plus une vue de taille 1. Par conséquent, tous les processus qui décident dans le comité r décident la même valeur.

Dans le cas contraire ($|view_i| > 1$), p_i met à jour sa proposition avant d'essayer de décider dans le comité suivant. Pour ce faire, il choisit la plus petite valeur dans $view_i$. Grâce à ce mécanisme simple, le nombre de propositions diminue strictement à chaque ronde. De plus, par hypothèse, les processus démarrent les rondes avec au plus k propositions. Par conséquent, au plus une valeur est proposée à la ronde k. Tout processus qui participe à cette ronde obtient donc une vue $view_i$ de cardinal 1 et décide.

Notation Dans la démonstration, nous utilisons les notations suivantes :

- var_i^r désigne la valeur de la variable locale var_i du processus p_i après modification éventuelle au cours de la ronde r.
- $PROP[r], 1 \leq r \leq k$ est l'ensemble des propositions au début de la ronde r. Plus précisément, $prop \in PROP[r]$ si et seulement si il existe un processus qui exécute $PS_{n,n}[r]$.participate$(id, < prop >)$ (ligne 3).

– $PROP[0]$ est l'ensemble des valeurs proposées. On a par hypothèse $|PROP[0]| \leq k$.

Lemme 2.3 $\forall r \in [1,k] : (1)\ |PROP[r]| \leq k+1-r$ *et* $(2)\ PROP[r] \subseteq PROP[r-1]$.

Démonstration La démonstration de la première partie du lemme procède par récurrence sur r. Soit $HR(r)$ la propriété $|PROP[r]| \leq k+1-r$.

– $HR(0)$
Par définition, $PROP[0]$ est l'ensemble des valeurs proposées. La spécification requiert que le nombre de valeurs proposées soit borné par k, i.e, $|PROP[0]| \leq k$.

– Soit $r \in [0, k-1]$. $HR(r) \Rightarrow HR(r+1)$
Supposons que $HR(r)$ est vraie. Nous commençons par démontrer que les vues $view_i$ calculées lors de la ronde r sont totalement ordonnées par inclusion. Nous remarquons ensuite que ces vues sont incluses dans $PROP[r]$ et que pour chaque niveau $\ell = |view_i| \geq 2$, au plus une valeur peut être choisie par les processus pour être proposée à la ronde suivante. Finalement, l'hypothèse de récurrence permet de conclure.

1. $\forall i,j : view_i^r \subseteq view_j^r \vee view_j^r \subseteq view_i^r$
Soit $v \in view_i$. p_i et p_j calculent leur vue à partir des ensembles S_i et S_j qu'ils obtiennent en appelant $PS_{n,n}[r].\mathsf{participate}(< id, prop >)$ (lignes 3-4). Ces ensembles sont ordonnés par inclusion. Sans perte de généralité, supposons que $S_i \subseteq S_j$. Il existe donc un couple $< id, v > \in S_i$, d'où $< id, v > \in S_j$. Par conséquent $v \in view_i$. Nous en concluons que $view_i \subseteq view_j$.
2. $\forall i : view_i^r \subseteq PROP[r]$
Cette propriété découle directement du code et de la définition de $PROP[r]$.
3. $\forall i,j : |view_i^r| = |view_j^r| \geq 2 \Rightarrow prop_i^r = prop_j^r$
Si $|view_i^r| = |view_j^r|$ alors $view_i^r = view_j^r = view$ (propriété 1 ci dessus). D'après le code, p_i et p_j choisissent la même valeur dans $view$ pour mettre à jour leur variable (ligne 5, la valeur choisie est la plus petite dans l'ensemble $view$). Nous avons donc $prop_i^r = prop_j^r$.
4. $|PROP[r+1]| \leq k+1-(r+1)$
Les vues $view_i^r$ sont totalement ordonnées par inclusion et incluses dans $PROP[r]$ (propriétés 1 et 2). Le nombre de vue est donc au maximum $|PROP[r]|$. Seules les vues de cardinal $\ell \geq 2$ génèrent une valeur dans $PROP[r+1]$ (ligne 5). De plus, pour $\ell \geq 2$ fixé, tous les processus p_i tels que $|view_i^r| = \ell$ ont la même valeur $prop$ à la fin de la ronde r (propriété 3). On en déduit que $|PROP[r+1]| \leq |PROP[r]|-1$, d'où, par $HR(r)$ $|PROP[r+1]| \leq k+1-(r+1)$.

Ceci termine la preuve de la partie (1) du lemme. Pour la deuxième partie, soit $r \in [1,k]$. Montrons que $PROP[r] \subseteq PROP[r-1]$. Soit v une valeur appartenant à $PROP[r]$. Par définition des ensembles $PROP$, il existe un processus p_i tel que $prop_i^{r-1} = v$. D'après le code, v est dans la vue $view_i$ calculée par p_i des valeurs proposées à la ronde $r-1$, i.e., $v \in PROP[r-1]$. On en déduit que $PROP[r] \subseteq PROP[r-1]$. $\square_{Lemme\ 2.3}$

Proposition 2.4 *L'algorithme décrit dans la Figure 2.13 est une implémentation sans attente d'un objet $BG_{[k],k}$ dans le modèle à mémoire partagée.*

Démonstration

Validité : Soit (r, d) la paire décidée par le processus p_i. D'après le code, d est l'unique valeur dans la vue $view_i$ calculée par p_i lors de la ronde r. On a donc $d \in PROP[r]$, d'où $d \in PROP[0]$ (Lemme 2.3). Finalement, par définition de l'ensemble $PROP[0]$, il existe un processus qui invoque $BG_{[k],k}.\mathsf{propose}(d)$.

Terminaison : Soit p_i un processus correct qui participe. Les objets de base $PS_{n,n}[1]$, ..., $PS_{n,n}[k]$ sont sans attente. De plus, le Lemme 2.3 montre qu'au plus une valeur est proposée à la ronde k. Si p_i ne décide pas lors des rondes $1, \ldots, k-1$, alors il décidera lors de la ronde k. En effet, on alors $PROP[k] = \{prop_i^{k-1}\}$ et donc $|view_i^k| = 1$. Le prédicat de la ligne 6 est satisfait et p_i décide.

Accord : Soient p_i et p_j deux processus qui décident respectivement (r_i, d_i) et (r_j, d_j) telles que $r_i = r_j = r$. D'après le texte de l'algorithme, $|view_i^r| = |view_j^r| = 1$, $view_i^r = \{d_i\}$ et $view_j^r = \{d\}$. De plus, nous avons vu dans la démonstration du Lemme 2.3 qu'au cours d'une ronde, les vues calculées sont totalement ordonnées par inclusion. Il existe donc une valeur v telle que $view_i^r = view_j^r = \{v\}$, d'où $d_i = d_j$.

$\square_{Proposition\ 2.4}$

2.4.3 Restreindre le nombre de propositions : du (n, k)-comité-binaire vers le $([k+1], k)$-BG

À ce stade, nous disposons d'une solution sans attente au problème $([k], k)$-BG qui utilise uniquement des registres. Nous avons vu que si l'on dispose d'un « moyen extérieur » (i.e., un objet (n, k)-accord) pour réduire le nombre de valeurs proposées de n à k, alors nous obtenons immédiatement une implémentation d'un objet (n, k)-comité (cf. paragraphe 2.4.1). Dans ce paragraphe, nous montrons comment, à l'aide d'un objet (n, k)-comité-binaire, résoudre le $([k+1], k)$-BG. Rappelons que n est une borne sur le nombre de processus participants tandis que la notation $[k+1]$ désigne le nombre de valeurs initiales. Nous cherchons donc à construire un objet $([k+1], k)$-BG accessible par au plus n processus qui sont collectivement munis d'au plus $k+1$ valeurs distinctes. Pour ce faire, nous utilisons un objet (n, k)-comité-binaire.

Les processus démarrent maintenant avec collectivement $k+1$ valeurs. Il s'agit d'une part de répartir ces valeurs sur k comités, en déposant au plus une valeur par comité et d'autre part, de s'assurer que chaque processus choisisse un comité dans lequel une valeur a été déposée. Nous avons vu dans le paragraphe précédent (paragraphe 2.4.2) comment réaliser cela lorsque le nombre de valeurs initiales est borné par k.

Considérons l'algorithme décrit dans la Figure 2.13 et supposons que $k+1$ processus proposent collectivement $k+1$ valeurs. Comment se comporte l'algorithme dans ces situations ? En analysant ces exécutions, on espère découvrir de quelle façon utiliser la puissance additionnelle du $(k+1, k)$-comité-binaire dans le but de résoudre le

$([k+1],k)$-BG. On remarque que dans certains cas, en fonction de l'étalement des valeurs sur les marches, l'algorithme fonctionne correctement. Par exemple, l'exécution schématisée dans la Figure 2.14 où $k=3$ produit des sorties légales. La figure décrit la répartition sur les marches dans chaque objet (n,n)-participating-set associé à chaque comité. Ainsi, dans l'objet associé au premier comité, deux vues $view_i$ sont calculées par les processus p_i : $vue_1 = \{1,2,3,4\}$ et $vue_2 = \{2,3,4\}$. Par conséquent, seules les valeurs $1 = \min(vue_1)$ et $2 = \min(vue_2)$ sont soumises au comité suivant. Deux valeurs sont proposées pour les deux comités restants : l'algorithme, restreint à ces deux comités se comporte alors correctement (dans l'exemple, 2 est décidé dans le comité 2 et 1 dans le comité 3).

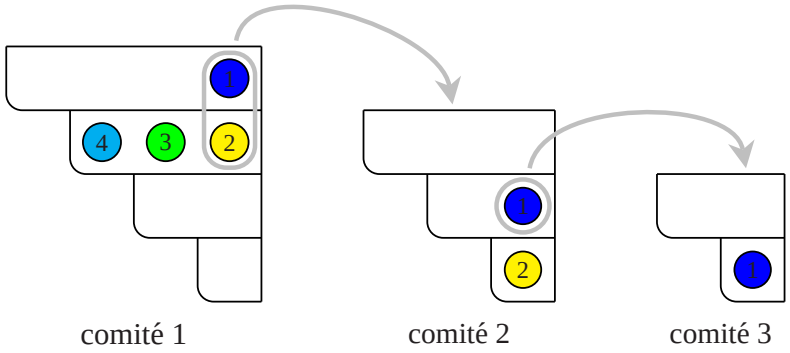

FIG. 2.14 – $BG_{[3],3}$, 4 valeurs initiales : une exécution valide

Par contre, la Figure 2.15 décrit une exécution dans laquelle certains processus ne décident pas. Dans le dernier comité, la plus petite vue est de cardinal 2. Les processus qui atteignent ce comité ne peuvent donc pas décider. Un examen attentif des « bonnes » exécutions révèle que la condition suivante est nécessaire à la mise en défaut de l'algorithme : dans chacun des objets (n,n)-participating-set, tout niveau $\ell > 1$ est non vide (la condition n'est pas suffisante ; pour s'en convaincre, dans l'exécution décrite dans la Figure 2.15, il suffit d'inverser 1 et 3 dans la répartition des valeurs dans l'objet associé au premier comité). Intuitivement, si le niveau $\ell > 1$ n'est pas occupé dans l'objet associé au comité c, alors la différence entre les nombres de valeurs soumises aux comités c et $c+1$ est d'au moins 2. Puisque le nombre de valeurs initiales est $k+1$, cette perte d'au moins deux valeurs suffit pour « normaliser » la situation dans les α comités restant (i.e., le nombre de valeurs proposées en entrée de ces α comités est inférieur ou égal à α).

La condition ci-dessus implique que si un processus ne parvient pas à décider à l'issue des k rondes, alors il existe pour chaque comité c une vue de cardinal 2. Par exemple, dans la Figure 2.15, les vues $\{3,4\},\{2,3\}$ et $\{1,2\}$ sont respectivement associées aux comités 1, 2 et 3. D'autre part, on sait que si une valeur est décidée dans le comité c, alors elle apparaît nécessairement dans la vue de taille 2 associée. Ces remarques suggèrent comment utiliser un objet (n,k)-comité-binaire pour résoudre le $([k+1],k)$-

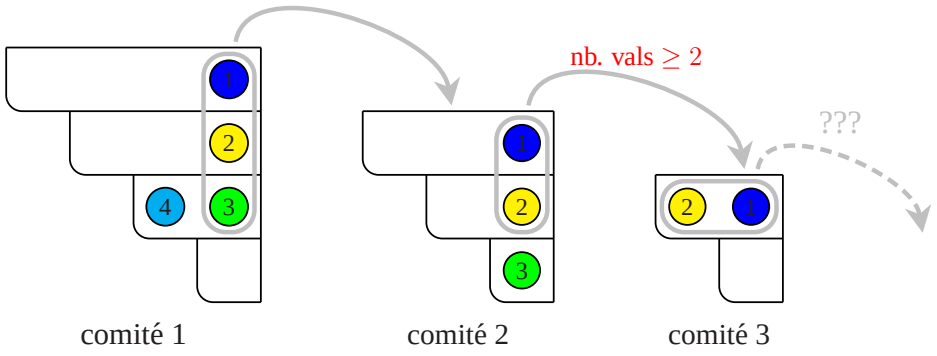

FIG. 2.15 – $BG_{[3],3}$, 4 valeurs initiales : mise en défaut de l'algorithme

BG. Intuitivement, dans les « mauvaises » exécutions, les processus soumettent à l'objet (n,k)-comité-binaire un vecteur binaire de taille k, chaque entrée c $(1 \leq c \leq k)$ encodant l'une des valeurs de la vue de taille 2 associée au comité c. En retour, chaque processus p_i obtient un couple $(c_i, d_i), d_i \in \{0,1\}$. Il décide alors la valeur désignée par le bit d_i dans la vue de taille 2 qui correspond au comité c_i. Toute la difficulté réside dans la réconciliation des décisions effectuées par le biais de l'algorithme 2.13 (le parcours ronde après ronde des objets (n,n)-participating-set) avec celles obtenues via l'appel à l'objet (n,k)-comité-binaire.

Principe et description de l'algorithme L'algorithme de la Figure 2.16 construit un objet $([k+1],k)$-BG accessible par n processus proposant globalement au plus $k+1$ valeurs à partir de registres atomiques et d'un objet (n,k)-comité-binaire. Les processus accèdent à l'objet en invoquant $BG_{[k+1],k}$.propose(v) et retournent une paire (numéro de comité, valeur) en exécutant l'instruction *return* à la ligne 08 ou 22.

L'algorithme se divise en deux phases :

- Première phase (lignes 01-12).
  Cette première phase est similaire à la construction de la Figure 2.13. Par rondes successives, les processus essaient de décider dans les comités $1, \ldots, k$. À chaque ronde, le nombre de valeurs proposées diminue. À la différence de la construction de la Figure 2.13, les processus écrivent en mémoire partagée les vues $view_i$ calculées lors des rondes auxquelles ils participent (ligne 05). De plus, le prédicat qui autorise la décision dans le comité r est plus contraint. p_i décide dans le comité r s'il observe une vue de taille 1 pour ce comité et si aucune vue de taille 2 n'a été annoncée (ligne 07).
  Puisque le nombre initial de valeurs est au plus $k+1$, il est possible que deux valeurs soient proposées lors de la ronde k (voir Figure 2.15 ou le Lemme 2.3). Ainsi, à l'issue de cette phase, il est possible que certains processus n'aient pas décider (en exécutant *return* à la ligne 08).
- Deuxième phase (lignes 13-22).

```
init : SS[1..k][1..k] ← [⊥,...,⊥] × [⊥,...,⊥] ;

operation BG_[k+1],k .propose(v_i)
(01) prop_i ← v_i ;
(02) for r_i = 1 to k do
(03)     S_i ← PS_n,n[r_i].participate(< id_i, prop_i >) ;
(04)     view_i ← {prop_j :< id_j, prop_j >∈ S_i} ;
(05)     SS[r_i, |view_i|].write(view_i) ;
(06)     for ℓ = 1 to k do ss_i[ℓ] ← SS[r_i, ℓ] enddo ;
(07)     if (ss[1] ≠ ⊥) ∧ (ss[2] = ⊥)  then let v such that  ss_i[1] = {v} ;
(08)                                        return((r_i, v))
(09)                                  else let ℓ_nv ≥ 2 such that  ss_i[ℓ_nv] ≠ ⊥ ;
(10)                                        prop_i ← min(ss_i[ℓ_nv])
(11)     endif
(12) enddo
(13) for ℓ_i = 1 to k do
(14)     let v_m (resp. v_M) be the smallest value (resp. greatest) value in SS[ℓ_i, 2] ;
(15)     case  (v_m ∈ SS[ℓ_i, 1])  then aux_i[ℓ_i] ← 0
(16)           (v_M ∈ SS[ℓ_i, 1])  then aux_i[ℓ_i] ← 1
(17)           default             then aux_i[ℓ_i] ← 0 or 1 arbitrarily
(18)     endcase
(19) enddo
(20) (c_i, d_i) ← BCD_n,k.propose(aux_i) ;
(21) let v = max(SS[c_i, 2]) if d_i = 1, min(SS[c_i, 2]) otherwise ;
(22) return(c_i, v)
```

FIG. 2.16 – $([k+1], k)$-BG dans $\mathcal{SM}_{n,n-1}[BCD_{n,k}]$, (code pour p_i)

Pour les processus, il s'agit de décider, à l'aide de l'objet (n,k)-comité-binaire, une paire *cohérente* avec les décisions éventuellement obtenues lors de la première phase. Pour cela, chaque processus p_i prépare un vecteur binaire ($\in \{0,1\}^k$) aux_i qui sera le paramètre d'entrée de l'invocation de l'objet $BCD_{n,k}$ (ligne 20).
Observons que les processus démarrent la première phase avec au plus $k+1$ valeurs. De plus, le nombre de valeurs décroît strictement d'une ronde à la suivante. Supposons qu'au cours d'une ronde, le nombre de valeurs perdues (i.e., qui ne sont pas proposées lors de la ronde suivante) est au moins 2. Le nombre de rondes qui restent à effectuer devient alors suffisant pour que chaque processus parvienne à décider. Autrement dit, après cette ronde, tout se passe comme dans la construction du $([k],k)$-BG : le nombre de valeurs est inférieur ou égal au nombre de comités restants.
Ainsi, des processus exécutent la deuxième phase si et seulement si exactement 1 valeur est perdue lors de chaque ronde de la première phase. Enfin, remarquons que ceci implique qu'une vue de taille 2 soit postée pour chaque ronde lors de la première phase (voir Lemme 2.4 et Figure 2.15). Ces remarques justifient la construction du vecteur aux_i : un bit suffit pour identifier une valeur eventuellement décidée dans le comité r puisque cette valeur apparaît nécessairement dans la vue de taille 2 associée à cette ronde. Par exemple, $aux_i[r] = 1$ indique que la proposition de p_i pour le comité r est la plus grande valeur contenue dans la vue de taille 2 qui correspond au comité r. De même, si le résultat de l'invocation de l'objet $BG_{[k+1],k}$ est $(r,0)$, p_i décide dans le comité r la plus petite valeur qui apparaît dans la vue de taille 2 ($SS[r,2]$) postée lors de la ronde r (lignes 21-22). Afin que les décisions pour un même comité r soient cohérentes entre les deux phases, lorsqu'il existe une vue de taille 1 ($SS[r,1] = \{v\} \neq \perp$), p_i propose la valeur contenue dans cette vue pour le comité r (i.e., $aux_i[r] = 1$ si $v = \max(SS[r,2])$, 0 sinon – lignes 14-18).

Objets partagés La construction repose sur un objet (n,k)-comité, k objets (n,n)-participating-set (notés $PS_{n,n}[1],\ldots,PS_{n,n}[k]$) et un tableau SS de registres atomiques de taille $k \times k$. Comme observé précédemment (paragraphe 2.4.2), les objets $PS_{n,n}$ sont construits à partir de registres atomiques. Les entrées du tableau SS sont initialisées à $\perp$, une valeur par défaut qui n'appartient pas à l'ensemble $\mathcal{V}$.

Pour simplifier le code, chaque entrée du tableau SS est un registre à *écrivains multiples*. Lors de la ronde r, il est possible que plusieurs processus écrivent dans l'entrée $SS[r,\ell]$ (ligne 05). Or, dans le cadre que nous avons fixé, la mémoire partagée est constituée de registres à écrivain unique. Du point de vue de la calculabilité, ces deux modèles sont équivalents puisque l'on peut construire des registres à écrivains multiples à partir de registres à écrivain uniques [120]. De plus, nous verrons que pour chaque entrée, au plus une valeur est écrite. Autrement dit, tous les processus qui écrivent dans le registre $SS[r,\ell]$ écrivent la même valeur. Lorsque cette propriété est garantie dans toute exécution, il est facile d'implémenter de façon sans attente un registre MW à écrivains multiples à partir d'un tableau $R[1,\ldots,n]$ de registres à écrivain unique. Pour écrire v dans MW, le processus p_i écrit v dans la $i$ème entrée $R[i]$ du tableau R. Pour

lire la valeur courante de MW, p_i collecte les valeurs des registres $R[1], \ldots, R[n]$. Il obtient ainsi l'un des ensembles $\{\perp\}$, $\{\perp, v\}$ ou $\{v\}$ où v est la seule valeur écrite par les processus dans MW. S'il observe v alors la lecture de MW retourne v. Dans le cas contraire, $\perp$ est retourné.

Correction de l'algorithme Concernant la première phase, la démonstration reprend les notations de la preuve de la construction de la Figure 2.13 :

- var_i^r désigne la valeur de la variable locale var_i du processus p_i après modification lors dans la ronde r (au cours de la première phase).
- $PROP[r], 1 \leq r \leq k$ est l'ensemble des propositions au début de la ronde r. Plus précisément, $prop \in PROP[r]$ si et seulement si il existe un processus qui exécute $PS_{n,n}[r].\mathsf{participate}(id_i, < prop >)$ (ligne 03).
- $PROP[0]$ est l'ensemble des valeurs proposées. On a par hypothèse $|PROP[0]| \leq k+1$.

Comme nous l'avons observé dans la description de l'algorithme, la première phase est similaire à la construction d'un objet $([k], k)$-BG. Seuls l'écriture des vues en mémoire partagée et le prédicat de décision diffèrent. Cependant, le mécanisme d'élimination des valeurs d'une ronde à l'autre demeure. En conséquence, le Lemme 2.3 est toujours valide, si l'on prend en compte le fait que le nombre initial de valeurs est maintenant borné par $k+1$.

Lemme 2.3 $\forall r \in [1, k] : (1)\ |PROP[r]| \leq (k+1) + 1 - r = k + 2 - r$ et (2) $PROP[r] \subseteq PROP[r-1]$.

Nous démontrons d'abord que s'il existe un processus qui exécute la deuxième phase (lignes 13-22) alors une vue de taille 2 a été postée pour chaque ronde $r, 1 \leq r \leq k$ avant que ce processus ne démarre la deuxième phase (Lemme 2.4). Nous utiliserons ce lemme pour démontrer que l'objet construit satisfait la spécification du problème $([k+1], k)$-BG (Proposition 2.5).

Lemme 2.4 *Soit une exécution dans laquelle il existe un processus p_i qui décide en exécutant la ligne 22. Lorsque p_i commence la seconde phase (c'est-à-dire lorsque p_i démarre l'exécution de la boucle* ***for*** *à la ligne 13), on a $\forall r \in [1, k] : SS[r, 2] \neq \perp$.*

Démonstration Pour obtenir une contradiction, supposons que le lemme est faux. Il existe donc un processus p_i qui décide à la ligne 22 et un numéro de ronde $R, 1 \leq R \leq r$ tels que p_i n'observe pas une vue de taille 2 associée à la ronde R. Plus précisément, lorsque p_i lit $SS[R, 2]$ au cours de la seconde phase (ligne 14), $SS[R, 2] = \perp$. La lecture (atomique) de $SS[R, 2]$ par p_i s'effectue à un certain instant que l'on note τ. D'après le code, une seule valeur $\neq \perp$ peut être écrite dans le registre $SS[R, 2]$. Ceci implique que $\forall \tau' \leq \tau$, $SS[R, 2] = \perp$.

Comme p_i exécute la deuxième phase de l'algorithme, il a précédemment essayé de décider au cours de la première phase dans chacun des comités $r = 1, \ldots, k$. Nous montrons que p_i aurait dû décider lors cette phase : une contradcition. Nous considérons deux cas, en fonction de la valeur de R.

- $R = k$.
  D'après le Lemme 2.3, $|PROP[k]| \leq 2$. La vue $view_i^r$ calculée par p_i au cours de la ronde k est donc de taille 1 ou 2. Or nous avons supposé que lorsque p_i exécute la rond k, aucun processus n'a posté ou ne poste une vue de taille 2 (c'est-à-dire que lorsque p_i collecte les vues à la ligne 06, on a $SS[k, 2] = \perp$). On en déduit d'une part que p_i calcule une vue de taille 1 lors de cette ronde et d'autre part que le prédicat qui conditionne la décision est satisfait (linge 07). Par conséquent, p_i décide dans le comité k au cours de la première phase.
- $1 \leq R < k$.
  Le cardinal de l'ensemble des valeurs proposées $PROP[R]$ dans le comité R est borné par $k + 2 - R$ (Lemme 2.3). Les valeurs proposées dans le comité $R + 1$ sont choisies parmi les plus petites valeurs des vues de taille ℓ, $2 \leq \ell \leq k + 2 - R$. Or, nous supposons qu'avant l'instant τ, on a toujours $SS[R, 2] = \perp$, c'est-à-dire qu'il n'existe pas de vue de taille 2 associée au comité R. Par conséquent, avant l'instant τ, le nombre de propositions $|PROP[R + 1]|$ pour le comité $R + 1$ est au plus $k - R$. Par suite, avant l'instant τ, on a $|PROP[R + 2]| \leq k - R - 1$, ..., $|PROP[k]| \leq 1$. Ceci découle du fait qu'au moins une valeur est éliminée à l'issue de chaque ronde (voir la démonstration du Lemme 2.3).
  An particulier, p_i exécute la ronde k avant l'instant τ. Au cours de cette ronde, il calcule nécessairement une vue de taille 1 et aucune vue de taille 2 n'est postée en mémoire partagée car $|PROP[k]| \leq 1$. Par conséquent, p_i décide au cours de la première phase.

$\square_{Lemme\ 2.4}$

Proposition 2.5 *L'algorithme décrit dans la Figure 2.16 est une implémentation sans attente pour au plus n processus d'un objet $BG_{[k+1],k}$ dans le modèle à mémoire partagée muni d'un objet (n, k)-comité-binaire.*

Démonstration La terminaison sans-attente découle directement du texte du protocole et des propriétés de terminaison des objets sous-jacents.

Validité Soit p_i un processus qui décide la paire (ℓ, v). Si p_i décide dans la première phase (ligne 08), v est contenue dans une vue de taille 1. Si p_i décide à la fin la seconde phase de l'algorithme, v appartient à une vue de taille 2 d'après le Lemme 2.4 et la ligne 21. Dans les deux cas, v appartient à une vue postée lors d'une ronde r au cours de la première phase de l'algorithme.
D'après le code, les vues $view_i$ postées lors de la ronde r au cours de la première phase sont incluses dans l'ensemble $PROP[r]$. On a donc $v \in PROP[r] \subseteq PROP[0]$, où $PROP[0]$ est l'ensemble des valeurs proposées (Lemme 2.3).

Accord $\forall \ell, 1 \leq \ell \leq k : p_i$ décide (ℓ, v_i) et p_j décide $(\ell, v_j) \Rightarrow v_i = v_j$.
Lorsque qu'un processus p_i décide (ℓ, v) à la ligne 08 ou 22, nous écrivons « p_i décide dans le comité ℓ ». Nous montrons que pour chaque comité $\ell, 1 \leq \ell, \leq k$, au plus une valeur est décidée. Soit $D[\ell]$ l'ensemble des processus qui décident dans le comité ℓ. Nous considérons plusieurs cas en fonction de(s) la(les) ligne(s) à la (aux)quelle(s) les processus de $D[\ell]$ décident.

- $\forall p_i \in D[\ell] : p_i$ décide à la ligne 08.
  Grâce aux propriétés de l'objet $PS_{n,n}[\ell]$, les vues $view_i$ calculées lors de la ronde ℓ sont ordonnées par inclusion (voir aussi la démonstration du Lemme 2.3). Par conséquent, au plus une vue de taille 1 peut être calculée lors de la ronde ℓ. D'après le code (lignes 07-08), tous les processus $p_i \in D[\ell]$ décident donc la même valeur.
- $\forall p_i \in D[\ell] : p_i$ décide à la ligne 22.
  Dans ce cas, chaque processus $p_i \in D[\ell]$ obtient en retour de son invocation $BCD_{n,k}.\mathsf{propose}(aux_i)$ une paire (ℓ, d_i). La propriété d'accord de l'objet garantit que $\exists d \in \{0,1\}$ tel que $\forall p_i \in D[\ell] : d_i = d$. De plus, lorsque $p_i \in D[\ell]$ lit $SS[\ell, 2]$ à la ligne 14 ou 21, $SS[\ell, 2] \neq \perp$ (Lemme 2.4). On en déduit, d'après le code (ligne 22) et le fait qu'il existe d tel que $\forall p_i \in D[\ell] : d_i = d$, que tous les processus qui appartiennent à l'ensemble $D[\ell]$ choisissent la même valeur dans $SS[\ell, 2]$.
- $\exists p_i, p_j \in D[\ell] : p_i$ décide à la ligne 08 et p_j décide à la ligne 22.
  Soit B l'ensemble que des processus qui invoquent l'objet $BCD_{n,k}$ (un processus qui appartient à cet ensemble ne décide pas nécessairement dans le comité ℓ). Parmi les processus $\in B$, soit p_b le premier processus qui lit le registre $SS[\ell, 1]$. Soit τ l'instant de cette lecture. Si p_b observe une valeur v, tous les processus $\in B$ proposent v pour le comité ℓ (lignes 15-16). v est alors la seule valeur qui peut être décidée dans le comité ℓ à l'aide de l'objet $CD_{n,k}$.
  Supposons maintenant que p_b n'observe pas de vue de taille 1 ($SS[\ell, 1] = \perp$) associé au comité ℓ. Dans ce cas, aucun processus ne décide dans le comité ℓ à la ligne 08. Lorsque p_b lit à l'instant τ le registre $SS[\ell, 1]$, $SS[\ell, 2] \neq \perp$ (Lemme 2.4). Par conséquent, un processus qui par la suite, observe $SS[\ell, 1] \neq \perp$ observe aussi $SS[\ell, 2] \neq \perp$ et ne peut donc pas décider en exécutant la ligne 08.

$\square_{Proposition\ 2.5}$

2.4.4 Du (n, k)-comité-binaire vers le (n, k)-comité

Finalement, dans ce paragraphe, nous réduisons le problème du (n, k)-comité au problème du (n, k)-comité-binaire. Maintenant que nous disposons d'un solution au problème du $([k + 1], k)$-BG fondée sur l'utilisation objets (n, k)-comité-binaire, il est facile de construire la réduction recherchée.

Les n processus démarrent avec n vecteurs potentiellement deux à deux distincts. Si l'on sait réduire ce nombre à au plus k, le problème est résolu (cf. algorithme 2.12, paragraphe 2.4.1). Or, un objet $BG_{[k+1],k}$, dont nous venons de voir une implémentation à partir d'un objet (n, k)-comité-binaire, possède justement ce pouvoir de diminuer le nombre de valeurs. Pour passer de $k + 1$ à k vecteurs, les n processus invoque l'objet $BG_{[k+1],k}$ avec en paramètre leur vecteur initial. En réponse, chaque processus p_i reçoit une paire (c_i, v_i). Il ignore le numéro de comité c_i et adopte le vecteur v_i. Ainsi, pour résoudre le (n, k)-comité, les n vecteurs initiaux passent par une chaîne d'objets $BG_{[n],n-1}, BG_{[n-1],n-2}, \ldots, BG_{[k+1],k}$, chaque objet $BG_{[\alpha+1],\alpha}$ étant construit à partir d'un objet $BCD_{n,\alpha}$ (il est immédiat qu'un objet $BCD_{n,k}$ implémente un objet $BCD_{n,\alpha}$

$\forall \alpha \geq k$). Cet algorithme simple est décrit dans la Figure 2.17. La preuve de la correction de cette transformation découle facilement du code et de la spécification des objets $BG_{[\alpha+1],\alpha}$.

```
operation CD_{n,k}.propose(vec_i)
(1)  prop_i ← vec_i ;
(2)  for α = n to k + 1 by −1 do
(3)      (c_i, prop_i) ← BG_{[α],α−1}.propose(prop_i)
(4)  enddo ;
(5)  return(c_i, prop_i[c_i])
```

FIG. 2.17 – (n,k)-comité dans $\mathcal{SM}_{n,n-1}[BG_{[n],n-1}, \ldots, BG_{[k+1],k}]$ (code pour p_i)

2.4.5 Du (n,k)-comité vers (n,k)-accord

L'algorithme est trivial. À partir de la valeur v_i qu'il souhaite proposer pour le (n,k)-accord, p_i forme un vecteur vec_i de taille k qui contient uniquement v_i. p_i invoque ensuite l'objet (n,k)-comité avec en paramètre vec_i. Finalement, p_i retourne la valeur décidée d_i obtenue par l'intermédiaire de l'objet (n,k)-comité. Le numéro de comité c_i est ignoré.

```
operation SA_{n,k} .propose(v_i)
(1)  vec_i ← [v_i, ..., v_i] ;
(2)  (c_i, d_i) ← CD_{n,k}.propose(vec_i) ;
(3)  return(d_i)
```

FIG. 2.18 – (n,k)-accord dans $\mathcal{SM}_{n,n-1}[CD_{n,k}]$ (code pour p_i)

La correction de l'algorithme est immédiate. Les propriétés de validité et terminaison découlent directement du code. Pour l'accord, le nombre de valeurs décidées dans chaque comité est au plus 1. De plus, le nombre de comités est égal à k. Le nombre de valeurs décidées collectivement est donc borné par k.

Résumé

Dans ce chapitre nous nous sommes d'abord intéressés aux relations entre renommage et accord « faible ». Nous avons établi une connexion entre ces deux schémas de coordination. Nous avons exhibé un compromis entre la « qualité » de l'accord et le cardinal de l'espace de renommage. En particulier, nous avons montré que les problèmes renommage adaptatif en $(2p - \lceil \frac{p}{k} \rceil)$ noms et $(k+1,k)$-test&set sont équivalents. C'est-à-dire que dans le modèle asynchrone, résoudre l'un de ces problèmes est aussi difficile que résoudre l'autre. Cette équivalence est montrée via une série de réductions algorithmiques sans attente (résumées dans la Figure 2.4, page 57).

Plus tard, il a été montré qu'il existe une algorithme sans attente qui implémente le renommage adaptatif en $g_k(p) = \min(2p-1, p+k-1)$ noms [53]. Cette transformation

est optimale : il n'est pas possible, à partir d'objet (n,k)-accord, d'obtenir un espace de noms plus petit de façon sans attente. Réciproquement, est-il possible de résoudre le (n,k)-accord à l'aide d'objet g_k-renommage adaptatifs ($g_k : p \to \min(2p-1, p+k-1)$) ? Dans [100], nous présentons pour le cas particulier $k=t$, un algorithme t-tolérant en mémoire partagée qui résout le problème du (n,k)-accord à l'aide d'objets g_k-renommage. Nous établissons également que pour certaines valeurs des paramètres k, t et n, une telle transformation est impossible. Les relations connues entre accord (test&set ou consensus ensembliste) et renommage sont décrites dans la Figure 2.19 (la flèche en pointillé signifie que la transformation est en général impossible).

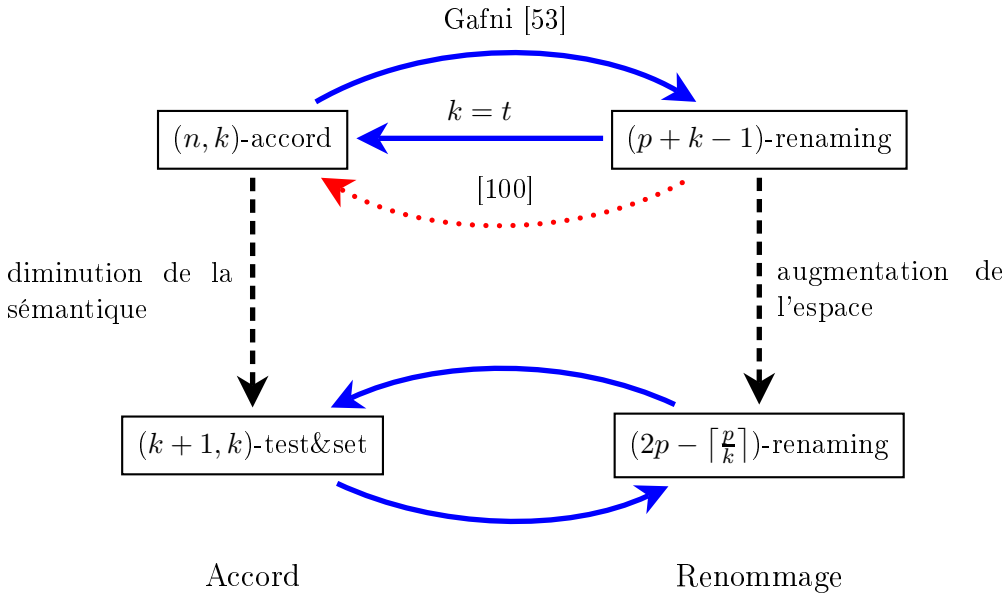

FIG. 2.19 – Réductions entre accord et renommage

La qualité de l'accord offerte par des objets test&set et consensus ensembliste diffère par la sémantique des valeurs décidées. Dans le problème test&set, il s'agit seulement d'un bit alors que le consensus ensembliste offre une plus grande richesse dans le sens où les valeurs décidées appartiennent à un ensemble qui peut être arbitrairement choisi. Nous observons que cette qualité sémantique influe directement sur la taille minimale de l'espace de renommage que l'on peut espérer obtenir.

Cette perspective ouvre plusieurs problèmes. Par exemple, étant donné une fonction qui détermine l'espace de renommage, peut-on définir un problème d'accord correspondant ? Comment unifier les relations dépeintes dans la Figure 2.19 ? Si l'on s'intéresse à la difficulté relative des problèmes de coordination faible, quels sont les paramètres pertinents (degré de tolérance aux défaillances, degré de « coordination » : k, taille de l'espace de renommage, etc., degré de concurrence ou nombre de processus n, ...) ?

Dans une deuxième partie, nous avons introduit un nouveau problème : la décision de comité. Ce problème capture l'idée d'essayer de mener simultanément des protocoles d'accord sur plusieurs fronts. Nous avons montré que la version binaire de ce problème

est équivalente au problème de l'accord ensembliste. Au contraire de [58] qui établit l'équivalence pour 3 processus, 2 comités entre les problèmes décision de comité et $(3, 2)$-accord à l'aide d'arguments topologique, l'approche adoptée ici repose exclusivement sur des réductions algorithmiques. Par contraste avec l'accord ensembliste qui consiste à choisir plusieurs valeurs, l'accord sur plusieurs fronts semble un problème plus naturel, qui pourrait avoir des applications pratiques. De plus, la décision en comités est plus riche : on peut imaginer plusieurs variantes, par exemple en autorisant la décision de plusieurs valeurs dans chaque comité ou en requérant que chaque processus décide dans plusieurs comités.

Chapitre 3

Composition de détecteurs de défaillances

Ce chapitre présente plusieurs résultats liés aux détecteurs de défaillances orientés vers la résolution du (n, k)-accord. Ces résultats ont été obtenus en collaboration avec Achour Mostéfaoui, Michel Raynal et Sergio Rajsbaum et ont fait l'objet d'une publication [92]. Le modèle choisi dans ce chapitre est le modèle asynchrone à passage de messages (noté $\mathcal{MP}_{n,t}[\emptyset]$).

Dans le chapitre précédent, nous avons étudiés des familles de problèmes de coordination faible aux contraintes relâchées. A partir d'un problème fortement contraint (e.g., le consensus), une famille de problèmes plus faibles est obtenue en relaxant l'une des propriétés de la spécification du problème initial (e.g., la famille des problèmes (n, k)-accord, $1 \leq k \leq n$). Ensuite, étant donné ces familles de problèmes de coordination faible, nous nous sommes intéressés aux relations les unissant du point de vue de la calculabilité. Nous avons abordé des questions du type suivant : étant donnée une solution au problèmes $P1$, existe-t-il un algorithme sans attente fondé sur cette solution qui résout $P2$? Les différentes réductions présentées ont permis de relier ces schémas de coordination en apparence de nature différente.

Nous adoptons la même démarche dans le contexte des détecteurs de défaillances orientés vers l'accord ensembliste. De même que la spécification du (n, k)-accord dérive de celle du consensus, plusieurs familles de détecteurs « faibles » ont été dérivées des spécifications de détecteurs qui offrent suffisamment d'information pour résoudre le consensus. Étant donné deux détecteurs arbitraires $\mathcal{C}_1$ et $\mathcal{C}_2$ dans ces familles, peut-on les comparer ? C'est à dire, existe-t-il un algorithme qui construit un détecteur de la classe $\mathcal{C}_2$ à partir d'un détecteur de la classe $\mathcal{C}_1$? Existe-t-il une hiérarchie entre ces classes de détecteurs ? Cette hiérarchie est-elle robuste ? Nous étudions des classes dérivées des détecteurs introduits dans le travail fondateur de Chandra, Hadzilacos et Toueg [30, 29], i.e., $\mathcal{S}/\diamond\mathcal{S}$, $\mathcal{P}/\diamond\mathcal{P}$ et Ω

Les détecteurs $\diamond\mathcal{S}$, Ω et $\mathcal{P}$ Pour circonvenir l'impossibilité du consensus dans un environnement asynchrone dans lequel les processus sont sujets aux pannes franches

[48], plusieurs classes de détecteurs de défaillances ont été proposées [29, 30]. En particulier, les classes $\mathcal{P}$, $\diamond\mathcal{S}$ et Ω introduites dans le premier chapitre (paragraphe 1.2) fournissent suffisamment d'informations sur les occurrences des défaillances pour résoudre le consensus. Par exemple, des algorithmes fondés sur la classe Ω sont proposés dans [29, 65, 81, 94, 117]. En d'autres termes, les hypothèses additionnelles de synchronie abstraites par chacun de ces oracles augmentent suffisamment la puissance de calcul du modèle pour résoudre le consensus. Deux résultats importants sont associés à ces classes. Premièrement, les classes $\diamond\mathcal{S}$ et Ω sont équivalentes et strictement plus faibles que la classe $\mathcal{P}$. C'est à dire qu'il existe un algorithme qui construit un détecteur de la classe Ω dans le modèle asynchrone enrichi avec un détecteur de la classe $\diamond\mathcal{S}$ [37, 28, 93] (la transformation inverse est triviale car $\Omega \subseteq \diamond\mathcal{S}$). D'autre part, s'il est évident que $\mathcal{P} \subseteq \diamond\mathcal{S}_x$ et donc qu'il est possible de construire un détecteur Ω à partir de $\diamond\mathcal{P}$, il n'est pas possible de construire un détecteur de la classe $\diamond\mathcal{P}$ à partir de $\diamond\mathcal{S}$ ou Ω [30, 28]. Deuxièmement, Ω (et donc $\diamond\mathcal{S}$) est la plus faible classe de détecteurs de défaillances qui permet de résoudre le consensus.

Les classes « mères » $\diamond\mathcal{S}$, Ω et $\mathcal{P}$ ont été affaiblies dans la littérature pour donner naissance aux familles $(\diamond\mathcal{S}_x)_{1\leq x\leq t+1}$, $(\Omega^z)_{1\leq z\leq t+1}$ et $(\phi^y)_{0\leq y\leq t}$. Chacune de ces familles a été introduite dans un but différent.

Familles de détecteurs dérivés de $\diamond\mathcal{S}$, Ω ou $\mathcal{P}$ La forme affaiblie $\diamond\mathcal{S}_x$ du détecteur $\diamond\mathcal{S}$ a été proposée dans [66] et étudiée du point de vue de la résolution du consensus ensembliste dans [95]. Des algorithmes de (n,k)-accord fondés sur la classe $\diamond\mathcal{S}_x$ sont donnés dans [71, 96, 122]. Contrairement à un détecteur $\diamond\mathcal{S}$ dont la portée s'étend tout le système (chaque processus reçoit une information pertinente du détecteur), la portée d'un détecteur $\diamond\mathcal{S}_x$ est restreinte à un sous-ensemble du système. La classe $\diamond\mathcal{S}_x$ est définie par la même propriété de complétude que la classe $\diamond\mathcal{S}$ (à partir d'un certain temps, la liste produite par le détecteur ne contient que des identités de processus corrects) et la propriété de précision *limitée* suivante : il existe un processus correct à qui, à partir d'un certain temps, au moins x processus font toujours confiance. Il est aisé de vérifier que $\diamond\mathcal{S}_n$ est $\diamond\mathcal{S}_x$ tandis que $\diamond\mathcal{S}_1$ n'apporte aucune puissance de calcul additionnelle. De plus, nous avons $\diamond\mathcal{S}_{x+1} \subseteq \diamond\mathcal{S}_x$. Par ailleurs, il a été montré qu'au sein de la famille $(\diamond\mathcal{S}_x)_{1\leq x\leq n}$, $\diamond\mathcal{S}_\alpha$ tel que $\alpha = t-k+2$ est la plus faible classe qui permet de résoudre le (n,k)-accord dans un système asynchrone [71] (le modèle à passage de message doit en sus satisfaire la contrainte $t < \frac{n}{2}$).

[90] étudie dans le cadre de la résolution du (n,k)-accord la combinaison des approches par condition [89] et détecteurs de défaillances. Dans ce but, Mostéfaoui et coauteurs définissent la famille $(\phi^y)_{0\leq y\leq n}$. Un détecteur ϕ^y offre une primitive QUERY(X) où le paramètre X désigne un ensemble de processus. Un appel QUERY(X) retourne une valeur booléenne qui dépend de l'état (défaillants ou vivants) des processus de X et de la taille de cet ensemble. QUERY(X) renvoie systématiquement *true* (resp. *false*) lorsque X est « trop petit » $(0 \leq |X| \leq t-y)$ (resp. lorsque X est « trop grand » ; $t < |X|$). Lorsque $t-y < |X| \leq t$, QUERY(X) retourne *true* si tous les processus de X sont défaillants. Clairement $\phi^{y+1} \subseteq \phi^y$ et ϕ^0 ne fournit aucune information sur les défaillances. Il est montré dans [90] qu'au sein de la famille $(\phi^y)_{0\leq y\leq n}$, ϕ^y est la plus

faible classe qui permet de résoudre le (n,k)-accord pour $k = t - y + 1$.

Enfin, Neiger [106] a introduit la famille $(\Omega^z)_{1\leq z\leq n}$ pour augmenter le rang des objets dans la hiérarchie de Herlihy. Étant donné un système équipé d'objets de rang $n-1$, il est montré dans [106] qu'il est possible de résoudre le $(n,1)$-accord à l'aide d'un détecteur Ω^{n-1}. [106] conjoncture également que Ω^z est la plus faible classe qui permet d'augmenter la puissance de synchronisation des primitives de rang z dans la hiérarchie. Ce résultat sera établi dans [63]. Un détecteur Ω^z fournit à chaque processus un ensemble d'au plus z identités de processus. À partir d'un certain temps, sur tous les processus le détecteur se stabilise sur le même ensemble qui contient l'identité d'au moins un processus correct. De même que pour les familles précédemment mentionnées, nous avons trivialement $\Omega^z \subseteq \Omega^{z+1}$, Ω^1 est Ω et la classe Ω^n ne fournit aucune information sur les défaillances.

Contributions Nous étudions les relations entre familles de détecteurs $(\mathcal{S}_x/\diamond\mathcal{S}_x)_{1\leq x\leq n}$, $(\phi^y/\diamond\phi^y)_{0\leq y\leq n-1}$ ([1]) et $(\Omega^z)_{1\leq z\leq n}$. Étant donné que nous savons que le (n,k)-accord peut être résolu à l'aide de :

- $\diamond\mathcal{S}_x$, $x = t - k + 2$;
- $\diamond\phi^y$, $y = t - k + 1$ et
- Ω^z, $z = k$ comme montré dans le paragraphe 3.3

il est naturel de s'interroger sur les puissances relatives de ces détecteurs. Plus précisément, ce chapitre étudie les question suivantes :

> *Les classes Ω^z, $\diamond\phi^y$ ou $\diamond\mathcal{S}_x$ qui offrent suffisamment d'information pour résoudre le (n,k)-accord sont-elles équivalentes ? Parmi ces détecteurs, quels sont les réductions possibles en fonction des paramètres x, y, z et t ? Le paramètre k étant fixé, quel est le plus faible détecteur au sein de ces familles qui offre suffisamment d'information pour résoudre le (n,k)-accord ? Quelle est la hiérarchie - si elle existe - qui les unie ? Cette hiérarchie est-elle robuste ? Ou est-il possible d'utiliser la puissance combinée de plusieurs détecteurs trop faibles individuellement pour résoudre le (n,k)-accord pour résoudre ce problème ? Dans ce cas, quel est la classe produite par ce type de combinaison ? Etc.*

La première contribution présentée dans le paragraphe 3.2 concerne les classes $\phi^y/\diamond\phi^y$. Dans le papier séminal de Chandra et Toueg [30] qui introduit le formalisme des détecteurs de défaillances, la sortie de ces oracles à un instant donné dépend uniquement du motif des occurrences des défaillances (voir paragraphe 1.2). Ce cadre contraste avec la définition de la famille $(\phi^y)_{0\leq y\leq n-1}$ puisque l'interrogation du détecteur implique un échange d'informations entre le processus et son module détecteur local par l'intermédiaire d'appels QUERY(X).

- Contribution # 1 : les classes $(\psi^y)_{0\leq y\leq n-1}$ et $(\diamond\psi^y)_{0\leq y\leq n-1}$.
  Nous définissons deux nouvelles familles de détecteurs de défaillances dont la sortie dépend uniquement des instants auxquels les processus consultent le détecteur

[1] $\diamond\mathcal{C}$ est la version « inéluctable » de $\mathcal{C}$. Dans toute exécution, les propriétés de $\mathcal{C}$ sont vérifiées à partir d'un certain temps a priori inconnu.

et du motif des défaillances courant. Ces détecteurs sont assez naturels car ils fournissent un entier qui est une approximation du nombre de processus défaillants. La précision de cette approximation dépend du nombre de processus qui sont défaillants au moment de l'interrogation et du paramètre y.

Plus précisément, l'entier renvoyé par un détecteur de la classe ψ^y est toujours compris entre $t-y$ et f, le nombre de processus défaillants dans l'exécution en cours. Soit f^τ le nombre de processus défaillant à l'instant τ. La seconde propriété de l'estimation peut s'exprimer ainsi : pour tout instant τ, il existe un instant $\tau' \geq \tau$ tel que à partir de cet instant, le détecteur renvoie un entier $\geq f^\tau$. La classe $\diamond\psi^y$ est simplement la version inéluctable de la classe ψ^y (l'entier renvoyé par un détecteur de cette classe satisfait les propriété de la classe ψ^y à partir d'un certain instant fini inconnu des processus ; avant cet instant, le comportement du détecteur est arbitraire).

Nous établissons dans le paragraphe 3.2 que les classes ψ^y et $\diamond\psi^y$ sont respectivement équivalentes aux classes ϕ^y et $\diamond\phi^y$.

Dans le reste du chapitre, nous étudions les réductions entre les détecteurs des familles $(\mathcal{S}_x/\diamond\mathcal{S}_x)_{1\leq x\leq n}$, $(\phi^y/\diamond\phi^y)_{0\leq y\leq n-1}$ et $(\Omega^z)_{1\leq z\leq n}$ ainsi que leur capacité à résoudre le (n,k)-accord pour k fixé. Dans ce qui suit, la notation $\mathcal{A}+\mathcal{B}\rightsquigarrow\mathcal{C}$ signifie que dans le modèle asynchrone équipé de détecteurs $\mathcal{A}$ et $\mathcal{B}$, il existe un algorithme qui construit un détecteur de la classe $\mathcal{C}$. De même, nous écrirons $\mathcal{A}+\mathcal{B}\not\rightsquigarrow\mathcal{C}$ lorsque cette transformation est impossible. Les notations $\mathcal{A}\rightsquigarrow\mathcal{C}$ et $\mathcal{A}\not\rightsquigarrow\mathcal{C}$ ont la même signification, en considérant un système équipé d'un détecteur de la seule classe $\mathcal{A}$.

- Contribution # 2 : Réductions.
  - Les classes $\mathcal{S}_x/\diamond\mathcal{S}_x$ et $\psi^y/\diamond\psi^y$ sont incomparables :
    - Soient $1\leq x\leq t+1$ et $1\leq y\leq t$. $\mathcal{S}_x\not\rightsquigarrow\diamond\psi^y$ (Théorème 3.4) ;
    - Soient $0\leq y\leq t-1$ et $2\leq x\leq t+1$. $\psi^y\not\rightsquigarrow\diamond\mathcal{S}_x$ (Théorème 3.5).
  - Réductions entre $\psi^y/\diamond\psi^y$ et Ω^z :
    - $\diamond\psi^y\rightsquigarrow\Omega^z$ si et seulement si $y+z>t$ (Corollaire 3.5) ;
    - Soient $1\leq z\leq t+1$ et $1\leq y\leq t$. $\Omega^z\not\rightsquigarrow\diamond\psi^y$ (Corollaire 3.7).
  - Réductions entre $\mathcal{S}_x/\diamond\mathcal{S}_x$ et Ω^z :
    - $\diamond\mathcal{S}_x\rightsquigarrow\Omega^z$ si et seulement si $x+z>t+1$ (Corollaire 3.6) ;
    - Soient $2\leq z\leq t+1$ et $2\leq x\leq t+1$. $\Omega^z\not\rightsquigarrow\diamond\mathcal{S}_x$ (Corollaire 3.8).

  Ces résultats sont résumés dans la Figure 3.1. Une flèche pleine joignant la classe $\mathcal{A}$ à la classe $\mathcal{B}$ indique qu'un détecteur de la classe $\mathcal{A}$ peut être transformé en un détecteur de la classe $\mathcal{B}$. Au contraire, une flèche en pointillés indique qu'une telle transformation est impossible. Les classes équivalentes sont encerclées par un cadre gris.

  Il est possible de classer ces détecteurs en fonction du plus petit k pour lesquelles elles fournissent suffisamment d'informations pour résoudre le (n,k)-accord. La colonne sur la droite matérialise ce classement. Toutes les détecteurs situés dans le $z$ième plan permettent de résoudre le (n,z)-accord. De plus, au sein d'une même famille définie par une colonne, la classe située sur le $z$ième plan est la plus faible classe qui permet de résoudre le (n,z)-accord. Enfin, parmi les classes situées sur un même « niveau z » dans la figure, la classe Ω^z est la plus faible classe pour

résoudre le (n, z)-accord.

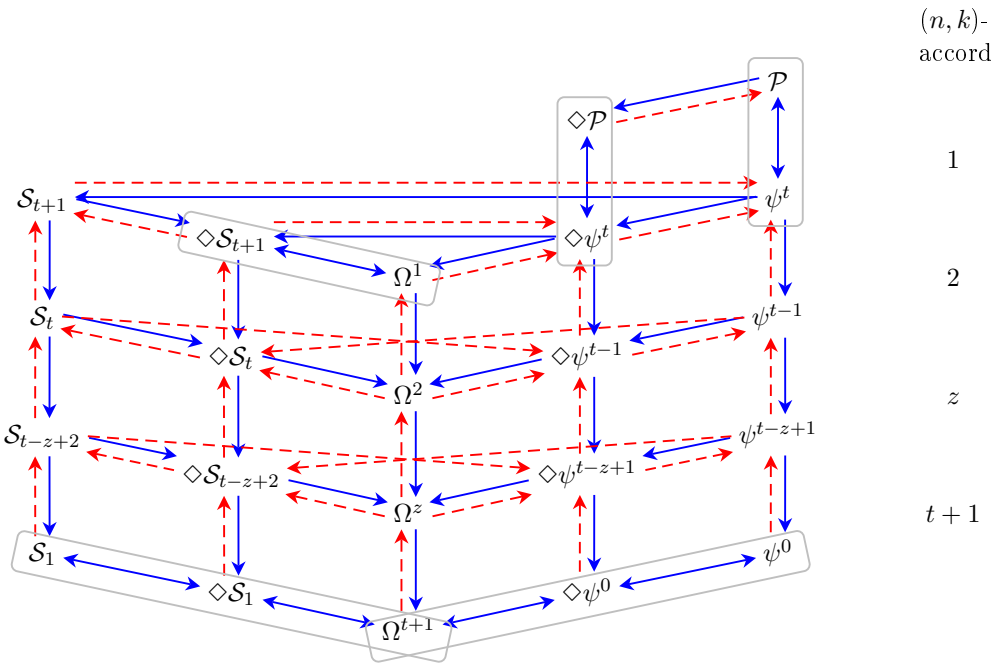

FIG. 3.1 – Grille de détecteurs de défaillances

– Contribution # 3 : Addition.
Par exemple dans le cas $t > 1$, considérons la classe $\Diamond\mathcal{S}_t$ qui permet de résoudre le $(n, 2)$-accord (mais qui ne permet pas de résoudre le consensus) et la classe $\Diamond\psi^{t-1}$ qui permet de résoudre le (n, k)-accord à condition que $k \geq t$. Du point de vue du (n, k)-accord, quelle la puissance de calcul que l'on obtient en combinant ces deux classes ? Nous montrons dans le paragraphe 3.4 que la combinaison $\Diamond\mathcal{S}_t + \Diamond\psi^{t-1}$ offre une puissance suffisante pour résoudre le consensus. Plus généralement, parmi les classes de la Figure 3.1 nous caractérisons les classes qui peuvent être additionnées et celles dont l'addition n'apporte aucune puissance de calcul supplémentaire. Ces résultats sont résumés par l'équation suivante : $\boxed{\Diamond\mathcal{S}_x + \Diamond\psi^y \rightsquigarrow \Omega^z \text{ si et seulement si } x + y + z > t + 1.}$ Pour établir ce résultat, nous présentons une construction algorithmique (Figures 3.7 et 3.8) et une preuve d'impossibilité (théorème 3.3).
Intuitivement, les informations fournies par les détecteurs $\Diamond\mathcal{S}_x$ et $\Diamond\psi^y$ sont de nature différente. Le gain obtenu par une telle addition est dépeint dans la Figure 3.2.
  – $\Diamond\mathcal{S}_x \rightsquigarrow \Omega^{t-x+2}$ et $\Diamond\mathcal{S}_x \not\rightsquigarrow \Omega^{t-x+1}$. Autrement dit, dans la famille $(\Omega^z)_{1\leq z\leq n}$, le détecteur le plus puissant qu'il est possible de construire à partir de $\Diamond\mathcal{S}_x$ est Ω^{t-x+2}. L'addition d'un détecteur de la classe $\Diamond\psi^y$ permet de passer de Ω^{t-x+2} à Ω^z avec $z = (t - x + 2) - y$.
  – De même $\Diamond\psi^y \rightsquigarrow \Omega^{t-y+1}$ et $\Diamond\psi^y \not\rightsquigarrow \Omega^{t-y}$. L'addition d'un détecteur de la classe

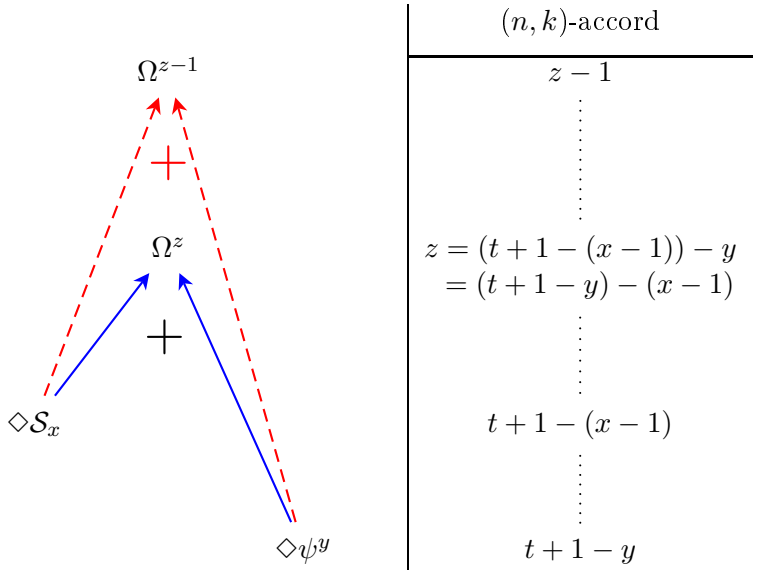

FIG. 3.2 – Addition des classe $\diamond\mathcal{S}_x$ et $\diamond\psi^y$

$\diamond\mathcal{S}_x$ permet d'obtenir un détecteur Ω^z tel que $z=(t-y+1)-(x-1)$.
Remarquons enfin que la hiérarchie dépeinte de la Figure 3.1 n'est pas robuste. En combinant deux détecteurs d'un même niveau z, il est possible d'atteindre un niveau $z'<z$ inférieur, c'est à dire d'obtenir une puissance de calcul supérieure. Par exemple, fixons $z=4$ et $t=5$. En additionnant les détecteurs des classes $\diamond\mathcal{S}_x$ et ψ^y du niveau 4 (c'est à dire $x=t-z+2=3$ et $y=t-z+1=2$), on obtient au mieux un détecteur de la classe $\Omega^{z'}$ avec $z'\geq t+2-x-y=2$.

– Contribution # 4 : Algorithme de (n,k)-accord fondé sur Ω^k.
Nous donnons un tel algorithme dans le paragraphe 3.3. L'algorithme proposé étend un protocole simple de consensus fondé sur la classe Ω. Nous montrons également que $k\geq z$ et $t<n/2$ est une condition nécessaire pour résoudre le (n,k)-accord dans le modèle asynchrone par passage de messages équipé d'un détecteur de la classe Ω^z (théorème 3.1). Ce théorème et la caractérisation des réductions possibles entre les différentes classes de détecteurs (Figure 3.1) implique que parmi les classes que nous étudions, Ω^z est la classe la plus faible qui permet de résoudre le (n,k)-accord.

Organisation du chapitre Les définitions des différentes familles de détecteurs sont données dans le paragraphe 3.1. L'équivalence entre les classes $\diamond\phi^y/\phi^y$ et $\diamond\psi^y/\psi^y$ est démontrée dans le paragraphe 3.2. Le paragraphe 3.3 présente un algorithme fondé sur Ω^z qui résout le (n,k)-accord pour $k\geq z$ et établit que dans la famille $(\Omega^z)_{1\leq z\leq n}$, Ω^k est la plus faible classe qui permet de résoudre ce problème. La construction d'un détecteur

Ω^z à partir de l'addition de $\diamond\mathcal{S}_x$ et $\diamond\psi^y$ est présentée dans le paragraphe 3.4. Enfin, les démonstrations l'optimalité de l'addition et des résultats d'impossibilités de la Figure 3.1 sont développées dans le paragraphe 3.5.

3.1 Une ménagerie de détecteurs de défaillances

Dans ce paragraphe, nous définissons formellement les familles de détecteurs brièvement présentées dans l'introduction. Chacune des classes d'une famille est paramétrée par un entier qui apparaît en indice ($\mathcal{C}_x$) ou en exposant ($\mathcal{C}^y$). L'indice indique la *portée* du détecteur, c'est à dire le nombre de processus qui obtiennent des informations pertinentes de l'oracle. L'exposant mesure la *qualité* de l'information donnée par l'oracle. Plus cet entier est petit, meilleur est la précision de l'oracle.

Pour le processus p_i, la sortie d'un détecteur $\mathcal{C}$ est matérialisée par une variable VAR_i accessible en lecture seule. La valeur de VAR_i est contrôlée par le détecteur $\mathcal{C}$. Ainsi, pour interroger le module local du détecteur, p_i lit le contenu de cette variable. Nous noterons VAR_i^τ la valeur de la variable à l'instant τ. Autrement dit, VAR_i^τ désigne la sortie du détecteur $\mathcal{C}$ pour le processus p_i à l'instant τ.

3.1.1 Les familles $(\diamond\mathcal{S}_x)_{1\leq x\leq n}$ et $(\mathcal{S}_x)_{1\leq x\leq n}$

Les classes de détecteurs de défaillances $\diamond\mathcal{S}_x$ et $\mathcal{S}_x$ ont été introduites et utilisées dans [66, 94, 96, 122]. Un détecteur $\diamond\mathcal{S}_x$ ou $\mathcal{S}_x$ fournit un ensemble d'identités de processus. Plus précisement, chaque processus p_i dispose d'une variable TRUSTED_i accessible uniquement en lecture dont la valeur est contrôlée par le détecteur sous-jacent. TRUSTED_i contient les identités des processus qui sont corrects du point de vue de p_i.

Un détecteur $\diamond\mathcal{S}_x$ satisfait les propriétés suivantes :

Définition 3.1 (Spécification de $\diamond\mathcal{S}_x$)

Complétude forte *Un processus défaillant est inéluctablement perçu comme tel :*
$\exists\tau, \forall i \in \Pi, \forall\tau' \geq \tau : \text{TRUSTED}_i^{\tau'} \subseteq \mathit{Correct}$

Faible précision à portée limitée *Un ensemble de processus de taille au moins x font, à un partir d'un certain instant, toujours confiance à un même processus correct :*
$\exists X \subseteq \Pi, \exists \ell \in \mathit{Correct} \cap X, \exists\tau : |X| \geq x \ \wedge \ (\forall\tau' \geq \tau : \ell \in \mathit{trusted}_i^{\tau'})$

La classe $\mathcal{S}_x$ généralise de façon similaire la classe $\mathcal{S}$. Un détecteur de la classe $\mathcal{S}_x$ satisfait la propriété de complétude forte énoncée ci-dessus ainsi que :

Définition 3.2 (Spécification de $\mathcal{S}_x$)

Faible précision à portée limitée, version perpétuelle *Un semble de processus de taille au moins x font toujours confiance à un même processus correct.*
$\exists X \subseteq \Pi, \exists \ell \in \mathit{Correct} \cap X : |X| \geq x \ \wedge \ (\forall\tau \geq 0 : \ell \in \text{TRUSTED}_i^\tau)$

Au sein de ces familles, plus la portée x est grande, plus le détecteur est puissant. De plus, un détecteur qui satisfait la précision faible de façon perpétuelle satisfait aussi la version non-perpétuelle de cette propriété. D'où $\mathcal{S}_x \subsetneq \diamond\mathcal{S}_x$, $\mathcal{S}_n \subseteq \ldots \subsetneq \mathcal{S}_{x+1} \subsetneq \mathcal{S}_x \subsetneq \ldots \subsetneq \mathcal{S}_1$ et $\diamond\mathcal{S}_n \subseteq \ldots \subsetneq \diamond\mathcal{S}_{x+1} \subsetneq \diamond\mathcal{S}_x \subsetneq \ldots \subsetneq \diamond\mathcal{S}_1$.

3.1.2 La famille $(\Omega^z)_{1\leq z\leq n}$

Un détecteur Ω^z maintient sur chaque processus un ensemble *leader* d'identités de processus de cardinal au plus z. La variable contrôlée par un tel détecteur est notée LEADER_i et satisfait la propriété suivante.

Définition 3.3 (Spécification de Ω^z)

- Leader multiple *Inéluctablement, tous les processus ont la même vision de l'ensemble des leaders. De plus, cet ensemble contient l'identité d'un processus correct.* $\exists L \subseteq \Pi,\ \exists \tau : (|L| \leq z)\ \wedge\ (L \cap Correct \neq \emptyset)\ \wedge\ (\forall \tau' \geq \tau, \forall i : \text{LEADER}_i^{\tau'} = L)$.

La famille $(\Omega^z)_{1\leq z\leq n}$ étend naturellement [106] la classe $\Omega(=\Omega^1)$ [29]. Une autre généralisation de Ω est étudiée dans [41]. Delporte-Gallet et coauteurs définissent la classe la classe Ω_S, où S est un sous ensemble prédéfini des processus du système. Un détecteur Ω_S requiert que tous les processus corrects $\in S$ s'accordent à partir d'un certain temps sur le même processus leader correct. Soit X l'ensemble de toutes les paires de processus. Les auteurs démontrent dans [41] que dans un système muni de tous les détecteurs $\Omega_x, x \in X$, il existe un algorithme qui construit un détecteur Ω.

3.1.3 Les familles $(\diamond\phi^y)_{1\leq y\leq n}$ et $(\phi^y)_{1\leq y\leq n}$

Ces classes ont été introduites dans [90] et [92]. Comme nous l'avons remarqué en introduction, leur définition ne rentre pas dans le cadre du travail fondateur sur les détecteurs de défaillances de Chandra et Toueg. En effet, la sortie d'un détecteur doit dépendre uniquement du motif des défaillances et de l'instant d'interrogation. Or, contrairement aux détecteurs précédemment définis qui fournissent une variable (TRUSTED ou LEADER) accessible en lecture seule, un détecteur $\diamond\phi^y$ ou ϕ^y exporte une primitive $\text{QUERY}(X)$, où le paramètre X choisi par le processus au moment de l'interrogation est un ensemble d'identités de processus. Cette primitive $\text{QUERY}(X)$ retourne une valeur booléenne. Elle permet au processus qui l'invoque de découvrir si une région du système est défaillante.

La famille $(\phi^y)_{1\leq y\leq n}$ Un détecteur de la classe ϕ^y satisfait les propriétés suivantes :

Définition 3.4 (Spécification de ϕ^y)

Trivialité *Si* $|X| \leq t-y$, $\text{QUERY}(X)$ *retourne* true. *Si* $|X| > t$, $\text{QUERY}(X)$ *retourne* false.

Sûreté *Si* $t-y < |X| \leq t$ *et au moins un processus* $p_i, i \in X$ *est vivant lorsque* $\text{QUERY}(X)$ *est invoquée alors le résultat de l'invocation est* false.

Vivacité *Soit* X *tel que* $t-y < |X| \leq t$. *Soit* τ *un instant tel que, à l'instant* τ, *tous les processus dont l'identité est dans* X *sont en panne. Il existe un instant* $\tau' \geq \tau$ *à partir duquel toute invocation de* $\text{QUERY}(X)$ *retourne* true.

Un détecteur ϕ^y offre une vision « grossière » du système. En effet, il n'est pas capable de rendre compte de l'état d'un groupe de $\alpha \leq t-y$ processus (propriété de trivialité). En particulier, pour $y < t$, la défaillance d'un unique processus n'est pas perçue par

un tel détecteur. Néanmoins, le détecteur indique de façon précise l'état des groupes X suffisamment larges ($|X| > t - y$). Si la réponse à un appel QUERY(X) est *true*, alors tous les processus de X sont défaillants (propriété de sûreté). De plus, pour un groupe X défaillants, il existe un instant à partir duquel toutes les invocations QUERY(X) signaleront que ce groupe est totalement défaillant (propriété de vivacité).

[90] montre que (1) $\phi^{y+1} \subseteq \phi^y$ et (2) les classes ϕ^t et $\mathcal{P}$ sont équivalentes dans le modèle asynchrone dans lequel au plus t processus sont susceptibles d'être défaillant au cours d'une même exécution. Enfin, il est aisé de vérifier qu'un détecteur ϕ^0 ne fournit aucune puissance de calcul additionnelle.

La famille $(\diamond\phi^y)_{1\leq y\leq n}$ Nous étendons la famille $(\phi^y_{1\leq y\leq n})$ en donnant une version « inéluctable ». Les propriétés sûreté et vivacité sont satisfaites à partir d'un certain instant, qui dépend de l'exécution et n'est pas connu des processus. Un détecteur de défaillance de la classe $\diamond\phi^y$ est défini par les propriétés suivantes :

Définition 3.5 (Spécification de $\diamond\phi^y$)

Trivialité *Si* $|X| \leq t - y$, QUERY(X) *retourne* true. *Si* $|X| > t$, QUERY(X) *retourne* false.

Sûreté inéluctable *Soit* X *tel que* $t - y < |X| \leq t$ *et* $X \cap Correct \neq \emptyset$. *Il existe un intant* τ *à partir duquel le résultat de toute invocation* QUERY(X) *à l'instant* τ' *est* false.

Vivacité *Soit* X *tel que* $t - y < |X| \leq t$ *et* $X \cap Correct = \emptyset$. *Soit* τ *un instant tel que, à l'instant* τ, *tous les processus de* X *sont défaillants. Il existe un instant* $\tau' \geq \tau$ *à partir duquel toute invocation de* QUERY(X) *retourne* true.

De même que pour la classe ϕ^y, les classes $\diamond\mathcal{P}$ et $\diamond\phi^t$ sont équivalente dans un système dans lequel au plus sont t processus sont fautifs au cours une même exécution. De plus, nous avons immédiatement $\diamond\phi^{y+1} \subseteq \diamond\phi^y$.

3.1.4 Les familles $(\diamond\psi^y)_{1\leq y\leq n}$ et $(\psi^y)_{1\leq y\leq n}$

La famille $(\psi^y)_{1\leq y\leq n}$ Un détecter ψ^y fournit à chaque processus une variable NB_C$_i$ accessible en lecture seule, qui contient un entier. Cet entier est une estimation du nombre de processus qui sont tombés en panne dans l'exécution courante (d'où le nom NB_C$_i$).

Rappelons que étant donné une exécution infinie, f ($0 \leq f \leq t$) dénote le nombre total de processus défaillant dans cette exécution. Ce nombre est bien défini car les pannes sont stables. Plus généralement, nous notons f^τ le nombre de processus qui sont tombés en panne avant l'instant τ. Un détecteur ψ^y satisfait les propriétés suivantes :

Définition 3.6 (Spécifications de ψ^y)

Sûreté : $\forall\tau : t - y \leq$ NB_C$_i^\tau \leq \max(t - y, f^\tau)$;

Vivacité : $\exists\tau : \forall\tau' \geq \tau,$ NB_C$_i^{\tau'} = \max(t - y, f)$.

La propriété de sûreté garantit que l'approximation donnée par l'oracle est toujours supérieure ou égale à $t - y$. De plus, lorsque le nombre de pannes dépasee le seuil

$t - y$, l'oracle ne sous-estime jamais le nombre de pannes. Enfin, dans ce cas, la sortie de l'oracle converge inéluctablement vers le nombre exact de défaillances (propriété de vivacité).

La famille $(\diamond\psi^y)_{1\leq y\leq n}$ La classe $\diamond\psi^y$ est la version inéluctable de la classe ϕ^y. La propriété de sûreté est relachée pendant une periode arbitraire mais finie. Cette propriété relâchée et la propriété de vivacité peuvent être rassemblées en une seule propriété :

Définition 3.7 (Spécifications de $\diamond\psi^y$)
Convergence inéluctable : $\exists\tau : \forall\tau' \geq \tau, \forall i \in \Pi : \text{NB\_C}_i^{\tau'} = \max(t - y, f)$.

Observons que, contrairement aux classe ϕ^y et $\diamond\phi^y$, les classes ψ^y et $\diamond\psi^y$ rentre dans le cadre défini par Chandra et Toueg. Le paragraphe suivant montre que ces deux familles sont équivalentes.

Notations (rappel) Soient $\mathcal{F}$ et $\mathcal{G}$ deux classes de détecteurs de défaillance. $\mathcal{MP}_{n,t}[\mathcal{F}]$ est utilisé pour représenter un système asynchrone de n processus parmi lesquels au plus t sont susceptibles de tomber en panne, qui communiquent par messages. Le système est enrichi avec un détecteur de défaillance de la classes $\mathcal{F}$. De même, la notation $\mathcal{MP}_{n,t}[\mathcal{F}, \mathcal{G}]$ représente un système équippé de détecteurs de défaillances des classes $\mathcal{F}$ et $\mathcal{G}$. Enfin, $\mathcal{MP}_{n,t}[\emptyset]$ désigne un système asynchrone « pure », c'est-à-dire sans détecteur de défaillances additionnel.

3.2 Les classes $\phi^y(\diamond\phi^y)$ et $\psi^y(\diamond\psi^y)$ sont équivalentes

Dans ce paragraphe, nous démontrons que les modèles $\mathcal{MP}_{n,t}[\phi^y]$ et $\mathcal{MP}_{n,t}[\psi^y]$ (respectivement, $\mathcal{MP}_{n,t}[\diamond\phi^y]$ et $\mathcal{MP}_{n,t}[\diamond\psi^y]$) offrent la même puissance de calcul. Pour ce faire, nous montrons comment construire un détecteur de la classe ψ^y ($\diamond\psi^y$) (respectivement ϕ^y ($\diamond\phi^y$)) dans le modèle $\mathcal{MP}_{n,t}[\phi^y]$ ($\mathcal{MP}_{n,t}[\diamond\phi^y]$) (respectivement $\mathcal{MP}_{n,t}[\psi^y]$ ($\mathcal{MP}_{n,t}[\diamond\psi^y]$)).

3.2.1 De $\phi^y(\diamond\phi^y)$ vers $\psi^y(\diamond\psi^y)$

Ce paragraphe décrit un algorithme simple qui construit un détecteur ψ^y (respectivement $\diamond\psi^y$) dans le modèle $\mathcal{MP}_{n,t}[\phi^y]$ (respectivement $\mathcal{MP}_{n,t}[\diamond\phi^y]$).

D'un coté, l'information fournie par la classe ψ^y ($\diamond\psi^y$) est une estimation du nombre de processus défaillants. D'un autre coté, pour un ensemble X, la classe ϕ^y ($\diamond\phi^y$) donne une information du type « il existe au moins un processus vivant dans X » ou « tous les processus de X sont défaillants ». Ainsi, pour trouver une approximation du nombre de processus défaillants, les processus testent tous les ensemble X possibles à l'aide de la primitive QUERY(X) fournie par le détecteur ϕ^y ($\diamond\phi^y$). L'approximation du nombre de défaillances est alors donnée par le cardinal du plus grand ensemble qui, selon le résultat des appels QUERY()), ne contient que des processus défaillants.

Brève description de la transformation L'algorithme de la Figure 3.3 implémente un détecteur de défaillances ψ^y (respectivement $\diamond\psi^y$) dans le modèle $\mathcal{MP}_{n,t}[\phi^y]$ (respectivement $\mathcal{MP}_{n,t}[\diamond\phi^y]$).

Pour chaque entier $\alpha \in [t-y+1, t]$, définissons $Sets(\alpha)$ comme l'ensemble qui contient tous les sous-ensembles de Π de cardinal α. Pour le processus p_i, l'algorithme consiste en une boucle infinie dont chaque itération met à jour la variable locale NB_C$_i$ qui simule la sortie courante du détecteur ψ^y (respectivement $\diamond\psi^y$).

A chaque itération de la boucle, p_i appelle QUERY(X) pour chaque ensemble X de cardinale α, α variant de $t-y+1$ à t. Si l'invocation QUERY(X) retourne *true* pour un ensemble X tel que $|X| = \alpha$, p_i en conclut que les α processus de X sont défaillants. Il inclut alors cette valeur dans l'ensemble A_i. Après avoir testé tous les ensembles possibles, p_i met à jour NB_C$_i$ en choisissant la plus grande valeur dans l'ensemble A_i.

```
init NB_C_i ← t − y

repeat forever
  A_i ← ∅;
  foreach α ∈ {t − y + 1, ..., t} do
      foreach X ∈ Sets(α) do
        if φ-QUERY(X) then A_i ← A_i ∪ {α} endif
      endfor
  endfor;
  if A_i ≠ ∅ then NB_C_i ← max(A_i) else NB_C_i ← (t − y) endif
endrepeat
```

FIG. 3.3 – De ϕ^y vers ψ^y (respectivement, de $\diamond\phi^y$ vers $\diamond\psi^y$), code pour p_i

Proposition 3.1 *L'algorithme de la Figure 3.3 implémente un détecteur de la classe ψ^y (respectivement $\diamond\psi^y$) dans le modèle $\mathcal{MP}_{n,t}[\phi^y]$ (respectivement $\mathcal{MP}_{n,t}[\diamond\phi^y]$).*

Démonstration La démonstration traite simultanément les cas du modèle $\mathcal{MP}_{n,t}[\phi^y]$ et le cas du modèle $\mathcal{MP}_{n,t}[\diamond\phi^y]$. Soit une exécution infinie arbitraire de l'algorithme. Nous étudions deux cas en fonction du nombre f de processus fautifs dans l'exécution considérée.

- $f < t-y+1$. Dans ce cas, $\forall\alpha \in [t-y+1, t], \forall X \in Sets(\alpha)$, il existe au moins un processus correct dans X. Donc, à partir d'un certain instant τ, les invocations ϕ-QUERY(X) retournent toujours *false* (propriété de sûreté du détecteur $\diamond\phi^y$ ou ϕ^y ; $\tau = 0$ dans le modèle $\mathcal{MP}_{n,t}[\phi^y]$ et $\tau \geq 0$ dans le modèle $\mathcal{MP}_{n,t}[\diamond\phi^y]$). Par conséquent, après τ, pour tout processus p_i, $A_i = \emptyset$ à la fin de chaque itération de la boucle externe. D'où, d'après le texte du protocole, l'existence d'un instant à partir duquel on a toujours, pour tout processus p_i, NB_C$_i = t-y = \max(t-y, f)$.
- $f \geq t-y+1$. Notons E l'ensemble des processus fautifs. On a $t \geq |E| \geq t-y+1$: il existe par définition des ensembles d'ensembles $Sets$, un entier $\alpha \in [t-y+1, t]$ tel que $E \in Sets(\alpha)$. De plus, d'après la propriété de vivacité de la classe $\diamond\phi^y$

ou ϕ^y, il existe donc un instant τ_1 à partir duquel les invocations ϕ-QUERY(E) retournent toujours *true*.
Observons que dans le modèle $\mathcal{MP}_{n,t}[\phi^y]$, avant l'instant τ_1, toutes les invocations ϕ-QUERY(E) retournent *false*. Au contraire, dans le modèle $\mathcal{MP}_{n,t}[\diamond\phi^y]$, une invocation ϕ-QUERY(E) avant τ_1 renvoie arbitrairement *true* ou *false*. Nous examinons deux cas en fonctions de la classe du détecteur de défaillances sous-jacent.

- Cas 1 : $\mathcal{MP}_{n,t}[\diamond\phi^y]$
  D'après la propriété de sûreté inéluctable, il existe un instant τ_2 à partir duquel les invocations ϕ-QUERY(X) telles que $|X| > |E| = f$ retournent *false*. Considérons une itération de la boucle externe qui débutent après l'instant $\tau = \max(\tau_1, \tau_2)$. A la fin de cette boucle, $f \in A_i$ et $\forall \alpha > f, \alpha \notin A_i$. On en conclut que inéluctablement, on a toujours NB_C$_i = f$: la propriété de convergence inéluctable de la classe $\diamond\psi^y$ est vérifiée.
- Cas 2 : $\mathcal{MP}_{n,t}[\phi^y]$
  Dans ce cas, en plus du fait que la valeur de NB_C$_i$ convergence vers f, il faut aussi établir qu'à tout instant la valeur de NB_C$_i$ est bornée par le nombre de processus effectivement défaillant à cet instant.
  Remarquons que le raisonnement du cas précédent s'applique également dans le modèle $\mathcal{MP}_{n,t}[\phi^y]$. La construction garantit donc la propriété de vivacité de la classe ψ^y (c'est à dire que la valeur de NB_C$_i$ converge vers f).
  Pendant la période durant laquelle le nombre de processus défaillants est inférieur à $t - y$, toutes les invocation ϕ-QUERY(X) telles que $t - y \leq |X| \leq t$ retournent *false*. Par conséquent, l'ensemble A_i est vide et on a NB_C$_i = t - y$ pendant cette période.
  Soient $\tau(f')$ le plus petit instant tel qu'il existe exactement f' $(1 \leq f' \leq f)$ processus défaillants. D'après la propriété de sûreté de la classe ϕ^y, les invocations ϕ-QUERY(X) telle que $|X| > f'$ retournent *false* au moins jusqu'à l'instant $\tau(f')$. Ainsi, la plus grande valeur contenue dans A_i jusqu'à $\tau(f')$ est bornée par f', ce qui montre que la construction garantit la propriété de sûreté de la classe ψ^y.

□*Proposition* 3.1

3.2.2 De $\psi^y(\diamond\psi^y)$ vers $\phi^y(\diamond\phi^y)$

Dans ce paragraphe, nous construisons un détecteur ϕ^y $(\diamond\phi^y)$ dans le modèle $\mathcal{MP}_{n,t}[\psi^y]$ $(\mathcal{MP}_{n,t}[\diamond\psi^y])$.

Principe de la transformation La construction d'un détecteur ϕ^y (resp. $\diamond\phi^y$) dans le modèle $\mathcal{MP}_{n,t}[\psi^y]$ (resp. $\mathcal{MP}_{n,t}[\diamond\psi^y]$) est décrite dans la Figure 3.4.

Lorsque p_i consulte le détecteur en invoquant ϕ-QUERY(X), il commence par vérifier si X est trop petit (ou trop grand). Si $|X| \leq t-y$ (respectivement $t < |X|$), l'algorithme retourne *true* (respectivement *false*). Autrement $(t-y < |X| \leq t)$, p_i essaie de déterminer si X contient au moins un processus vivant. Pour faire cela, p_i écrit la valeur courante

de NB_C$_i$ dans la variable locale est_c_i et diffuse un message Inquiry daté avec un entier sn_i (sn_i est le premier numéro de séquence non encore utilisé par p_i).

Il attend ensuite la réception d'un nombre suffisant de messages Response correspondant (i.e., qui transportent le numéro de séquence courant sn_i) ou le changement de la valeur de $n - $ NB_C$_i$ (ligne 06). « Un nombre suffisant » veut dire ici au moins $n - $ NB_C$_i$ (pour éviter de possibles blocages au cours de l'attente, p_i évalue régulièrement la condition de la ligne 06). Si la valeur de NB_C$_i$ a changé, p_i entame un nouveau cycle d'interrogation (lignes 04-06). Autrement, p_i collecte les identités des processus desquels il a reçu un message Response(sn_i) dans la variable rec_i. Enfin, si il existe un processus $p_j \in X$ tel que $j \in rec_i$ alors ce processus était vivant au début du cycle d'interrogation daté sn_i. Par conséquent, p_i retourne *false*. Dans le cas contraire, p_i retourne *true* (ligne 09).

```
operation φ-QUERY(X) :
(01) case |X| ≤ t − y          then return (true)
(02)      t < |X|              then return (false)
(03)      t − y < |X| ≤ t      then
(04)           repeat sn_i ← sn_i + 1 ; est_c_i ← NB_C_i ;
(05)                foreach j ∈ {1, ..., n} do send Inquiry(sn_i) to p_j enddo ;
(06)                wait until ( (Response(sn_i) received from n − est_c_i processes)
                                 ∨ (est_c_i ≠ NB_C_i) ) ;
(07)           until est_c_i = NB_C_i endrepeat ;
(08)           let rec_i = {j | RESPONSE(sn_i) has been received from p_j} ;
(09)           return (X ∩ rec_i = ∅)
(10) endcase

Background task : when Inquiry(sn) is received from p_j : send Response(sn) to p_j
```

FIG. 3.4 – De ψ^y vers ϕ^y (ou de $\diamond\psi^y$ vers $\diamond\phi^y$), code pour p_i

Proposition 3.2 *L'algorithme de la Figure 3.4 construit un détecteur de la classe ϕ^y (respectivement $\diamond\phi^y$) dans le modèle $\mathcal{MP}_{n,t}[\psi^y]$ (respectivement $\mathcal{MP}_{n,t}[\diamond\psi^y]$).*

Démonstration Considérons une exécution arbitraire de l'algorithme. f est le nombre de défaillances dans cette exécution. La démonstration se décomposent en cinq parties.

- [Terminaison] Montrons que toute invocation ϕ-QUERY(X) par un processus correct termine. Ceci est trivialement vérifié si $|X| \leq t-y$ ou $t < |X|$. Nous supposons donc que $t - y < |X| \leq t$.
  D'après la propriété de sûreté de la classe ψ^y ou de convergence inéluctable de la classe $\diamond\psi^y$, il existe un instant à partir duquel NB_C$_i = \max(t-y, f)$. Puisque la variable est_c_i est périodiquement mise à jour avec l'estimation fournie par le détecteur (ligne 04), il existe un instant τ à partir duquel on a toujours $est\_c_i = \max(t-y, f)$. D'où, à partir de τ, $n - est\_c_i \leq n - f = |Correct|$. De plus lorsqu'un processus correct reçoit un message Inquiry(sn) de p_i, il répond en envoyant un message Response(sn). Ainsi, pour un numéro de séquence donné sn, p_i reçoit

au moins $n - f$ messages Response(sn) correspondant. Par conséquent, l'attente (ligne 06) n'est pas bloquante. De même, le prédicat de la ligne 07 est à partir d'un certain temps toujours vérifié car, après τ, on a toujours $est\_c_i = \text{NB\_C}_i$.

- [Propriété de trivialité de ϕ^y ou $\Diamond\phi^y$] Cette propriété découle immédiatement du code (lignes 01–02).
- [Propriété de vivacité de ϕ^y ou $\Diamond\phi^y$] Soit E tel que $|E| > t - y$ un ensemble de processus défaillants. Soit $\tau(E)$ l'instant de la dernière défaillance dans E. Considérons une invocation ϕ-QUERY(E) qui démarre après τ. Cette invocation termine (cf. ci-dessus) et l'ensemble rec_i calculé à partir des messages Response(sn) reçus par p_i (ligne 08) ne contient pas d'identités de processus de E. En effet, seul p_i génère des messages Inquiry datés sn et l'envoi de ces messages débute après la dernière défaillance des processus de E. Nous avons donc $rec_i \cap E = \emptyset$: l'invocation retourne *true*.
- [Propriété de sûreté de la classe $\Diamond\phi^y$] Soit X un ensemble de processus tel que $t - y < |X| \leq t$ et $X \cap Correct \neq \emptyset$. Nous devons établir qu'il existe un instant à partir duquel toute invocation ϕ-QUERY(X) retourne *false*, c'est-à-dire qu'il existe un instant à partir duquel $rec_i \cap X \neq \emptyset$ est toujours vrai. D'après le code et la propriété de convergence inéluctable de la classe $\Diamond\psi^y$, il existe un instant τ à partir duquel on a toujours $est\_c_i = \text{NB\_C}_i = \max(t - y, f)$. Considérons un cycle d'envoi/réception de messages Inquiry/Response datés sn qui démarre après l'instant τ. La collecte des messages Response(sn) se termine lorsque p_i a reçu $n - \max(t - y, f)$ messages. D'après le code, nous avons alors $|rec_i| = n - \max(t - y, f)$. Si $f < t - y$ alors $rec_i \cap E \neq \emptyset$ car $|E| > t - y$. Sinon, $|rec_i| = n - f$ et rec_i est, au bout d'un certain temps postérieur à la défaillance des f processus fautifs, exactement l'ensemble des processus corrects. D'où, inéluctablement $rec_i \cap E \neq \emptyset$ car E contient l'identité d'un processus correct.
- [Propriété de sûreté de ϕ^y] Soient X un ensemble de processus et τ un instant tels que $t - y < |X| \leq t$ et l'un des processus de X est vivant à l'instant τ. Nous devons établir que toute opération ϕ-QUERY(X) terminée avant τ retourne *false*.
  - Cas 1 : $f \leq t - y$. Dans ce cas, dès l'instant initial nous avons toujours $\text{NB\_C}_i = t - y$ (propriété de sûreté de la classe ψ^y) d'où $est\_c_i = \text{NB\_C}_i = t - y$. Par conséquent, $|rec_i| = n - (t - y)$, d'où $rec_i \cap E \neq \emptyset$ car $|E| > t - y$. Toute opération ϕ-QUERY(X) renvoie donc *false*.
  - Cas 2 : $f > t - y$. Nous démontrons d'abord la propriété P suivante : lorsque qu'un processus p_i termine la boucle **repeat**, il a reçu un message Response de chaque processus vivant à l'instant où l'opération ϕ-QUERY(X) en cours termine.

    Considérons la dernière exécution du corps de la boucle **repeat** lors d'une opération ϕ-QUERY(X). Soient sn le numéro de séquence correspondant, τ_b l'instant auquel p_i lit la valeur courante (notée x) de NB_C_i (ligne 04) et τ_e l'instant auquel p_i obtient de nouveau x en lisant NB_C_i (ligne 07). En particulier, au cours de l'exécution de cette itération de la boucle, l'attente (ligne 06) se termine lorsque p_i a reçu $n - x$ messages Response(sn). Soient f^{τ_b} et f^{τ_e} le nombre de processus défaillants aux instant τ_b et τ_e respectivement.

D'après la propriété de sûreté de ϕ^y, nous avons $x \leq f^{\tau_b} \leq f^{\tau_e}$. De plus, (1) les processus défaillants à l'instant τ_b n'envoient pas de messages Response(sn) ; (2) tous les processus vivants à l'instant τ_e envoient un message Response(sn) à p_i ; (3) à la fin de l'itération, p_i a reçu $n-x$ messages Response(sn) ; et (4) $n-x \geq n-f^{\tau_e}$. Nous en concluons que p_i reçoit un message Response(sn) de chaque processus vivant à l'instant τ_e, ce qui démontre la propriété P.
Soit A l'ensemble des processus vivant lorsque l'opération ϕ-QUERY(X) termine. Nous venons de montrer que $A \subseteq rec_i$. Ainsi, si X contient un processus vivant lorsque l'opération ϕ-QUERY(X) termine, $X \cap A \neq \emptyset$, d'où $X \cap rec_i \neq \emptyset$. Il s'ensuit que ϕ-QUERY(X) retourne *false*.

□*Proposition* 3.2

Simplification de la transformation pour la classe $\diamond\phi^y$ La propriété de sûreté de la classe ϕ^y est obtenue au moyen d'un mécanisme de synchronisation fort réalisé par la boucle **repeat** et les variables locales est_c_i et sn_i (lignes 04-07). Ce mécanisme est utilisé pour isoler un cycle d'interrogation Inquiry/Response durant lequel NB_C$_i$ ne change pas.

Cette synchronisation n'est pas nécessaire pour garantir la propriété de sûreté inéluctable de la classe $\diamond\phi^y$. Plus précisément, nous pouvons supprimer les variables locales est_c_i an sn_i et remplacer la boucle **repeat** (lignes 04-07) par le code suivant :

for_each $j \in \{1,\ldots,n\}$ **do** *send* Inquiry(sn_i) *to* p_j **end_do**;
wait until (Response() received from $n-$ NB_C$_i$ processes).

Nous laissons la démonstration au lecteur.

3.3 (n,k)-accord dans le modèle $\mathcal{MP}_{n,t}[\Omega^z]$

Dans cette section, nous étudions la (im)possibilité de résoudre le problème (n,k)-accord (cf. paragraphe 2.1.1) dans le modèle $\mathcal{MP}_{n,t}[\Omega^z]$. Nous répondons à la question suivante : étant données les valeurs des paramètres k et z, existe-t-il un algorithme qui résout le problème du (n,k)-accord dans le modèle $\mathcal{MP}_{n,t}[\Omega^z]$?

1. Nous présentons un algorithme qui résout le problème du k-accord dès que le système asynchrone est equippé d'un détecteur de défaillances de la classe Ω^z tel que $z \leq k$. (paragraphe 3.3.1, proposition 3.3). L'algorithme appartient à la classe des protocoles indulgents [64, 65] : la sortie d'un tel protocole est toujours *sûre*, quelque soit le comportement du système.
2. Nous montrons ensuite que l'algorithme est, du point de vue de la calculabilité du (n,k)-accord dans la famille $(\Omega^z)_{1\leq z\leq n}$, optimal : si $k<z$, il n'existe pas d'algorithme solution au problème (n,k)-accord dans le modèle $\mathcal{MP}_{n,t}[\Omega^z]$ (paragraphe 3.3.3, théorème 3.1).

3.3.1 Un algorithme pour le (n, k)-accord

L'algorithme, décrit à la Figure 3.5, est une simple adaptation d'un algorithme de consensus fondé sur la classe Ω [65], lui même inspiré d'un algorithme de consensus fondé sur la classe $\diamond\mathcal{S}$ [94]. Il suppose l'existence d'une majorité de processus corrects ($t < \frac{n}{2}$). Pour le processus p_i, l'algorithme est lançé lorsqu'il invoque k-set_agreement(v_i), où v_i est la valeur qu'il propose. Si p_i ne tombe pas en panne, l'algorithme termine lorsque p_i exécute *return*(v). v est alors la valeur décidée par p_i.

L'algorithme se décompose en deux tâches $T1$ et $T2$ exécutées en parallèle. La tâche $T2$ dissémine une valeur décidée et évite les interblocages. Dans la tâche principale $T1$, les calculs s'effectuent par rondes asynchrones successives. Chaque ronde est divisée en deux phases, chaque phase incluant une étape de communication globale entre les processus. Pour le processus p_i, la variable est_i contient l'estimation courante de la valeur de décision et r_i est le numéro de la ronde courante.

Au cours de la première phase de la ronde r, p_i lit LEADER$_i$ (l'ensemble fourni par le détecteur Ω^z), mémorise sa valeur dans la variable L_i et envoie un message Phase1(r, L_i, est_i) à tous les processus. Ensuite, p_i attend la réception de $n-t$ messages Phase1 (c'est à dire d'une majorité) envoyés durant cette même ronde r. Il attend également la réception d'un message Phase1 de la ronde r émis par un processus p_j qu'il considère comme leader ($j \in L_i$) ou le changement de la valeur de LEADERS$_i$. Si une majorité de processus ont le même ensemble de leaders L et p_i a reçu une estimation v_L d'un processus appartenant à cet ensemble, il mémorise v_L dans la variable aux_i. Dans le cas contraire, aux_i est positionnée à $\perp$, où $\perp$ est une valeur spéciale qui n'est jamais proposée. La propriété cruciale de cette première phase est la suivante : la taille de l'ensemble des valeurs $\neq \perp$ contenues dans les variables locales aux_i est bornée par $|L_i| \leq z \leq k$ (voir la preuve de l'algorithme, Lemme 3.2). L'objectif de cette première phase est donc de réduire le nombre de valeurs dans le système à un total $\leq k$.

Le but de la seconde phase est de garantir que le nombre de valeurs décidées est inférieure ou égale à k, et ce quelques soient les rondes au cours desquelles des processus décident. La correction du mécanisme algorithmique repose sur l'hypothèse d'une majorité de processus corrects ($t < n/2$). Au cours de cette phase, chaque processus p_i diffuse un message Phase2(r_i, aux_i) à tous les processus puis attend la réception d'au moins $n - t$ messages Phase2 de la ronde courante. Si p_i a reçu une valeur $\neq \perp$ au cours de cet échange, il l'adopte comme nouvelle estimation de la valeur à décider (si differentes valeurs $\neq \perp$ ont été reçues, l'une d'entre elles est choisie de façon arbitraire). De plus, si aucune valeur $aux = \perp$ n'a été reçue p_i décide la valeur qu'il a précédemment adoptée. Toutefois, pour éviter le blocage des autres processus lors des étapes d'attente, p_i diffuse v à l'aide d'une primitive de *diffusion fiable* (cf. paragraphe 1.1) avant de décider (tâche $T2$).

3.3.2 Preuve de l'algorithme

La preuve est similaire à la preuve de l'algorithme de consensus décrite dans [65]. La correction de l'algorithme repose sur les hypothèses (1) $t < n/2$ et (2) $z \leq k$.

```
function k-set_agreement(v_i) :

init : est_i ← v_i ; r_i ← 0

task T1 :
(01)  repeat forever
        ———————— Phase 1 ————————————————————————————————
(02)    r_i ← r_i + 1 ; L_i ← LEADERS_i ;
(03)    broadcast Phase1(r_i, L_i, est_i) ;
(04)    wait until (Phase1(r_i, _, _) received from ≥ (n − t) processes) ;
(05)    wait until ( (Phase1(r_i, _, _) received from a process ∈ L_i) ∨ (L_i ≠ LEADER_i)) ;
(06)    if ((∃L :Phase1(r_i, L, _) received from a majority of processes)
(07)          ∧(Phase1(r_i, _, v_L) received from a process ∈ L))
(08)      then aux_i ← v_L else aux_i ← ⊥ end_if ;
          % Here |{aux_j : aux_j ≠ ⊥}| ≤ |L_i| ≤ k %
        ———————— Phase 2 ————————————————————————————————
(09)    broadcast Phase2(r_i, aux_i) ;
(10)    wait until (Phase2(r_i, _) received from (n − t) processes ) ;
(11)    let rec_i = { aux :Phase2(r_i, aux) has been received } ;
(12)    if (∃v : v ≠ ⊥ ∧ v ∈ rec_i) then est_i ← v end_if ;
(13)    if (⊥ ∉ rec_i) then R_Broadcast Decision(est_i) ; stop T1 end_if
(14)  end_repeat

task T2 : when Decision(v) is R_ delivered : return(v) ; stop T2
```

FIG. 3.5 – Algorithme de (n,k)-accord dans $\mathcal{MP}_{n,t}[\Omega^z]$; requiert $t < n/2$ et $z \leq k$

Lemme 3.1 *Aucun processus correct n'est bloqué au cours d'une ronde.*

Démonstration Soit p_i un processus correct. Nous devons montrer que, quelque soit le numéro de ronde r, les étapes d'attente de messages (lignes 04, 05 ou 10) se terminent.

- Un processus correct décide. Dans ce cas, un message Decision(v) est R_livré par ce processus (tâche $T2$). La propriété de terminaison de la diffusion fiable (définition 1.1) garantit que ce message est R_livré par tous les processus corrects. Par conséquent, tous les processus corrects décident (tâche $T2$) et ne sont donc pas bloqués de façon permanente lors des étapes d'attente.
- Il n'existe pas de processus correct qui décide. Soit r le plus petit numéro de ronde au cours de laquelle un processus correct que l'on notera p_i est bloqué. Lorsque $r_i = r$, p_i est bloqué à la ligne 04, 05 ou 10.
  1. p_i est bloqué ligne 04. Par définition de r, tous les processus corrects diffuse un message Phase1$(r, _, _)$. Ces messages seront reçus par p_i car les échanges de messages entre processus corrects sont fiables (paragraphe 1.1.2.1). Comme le nombre de défaillances est borné par t, il existe donc un instant à partir duquel le prédicat de la ligne 04 est satisfait pour p_i.
  2. p_i est bloqué ligne 05. Par définition de la classe Ω^z (définition 3.3), il existe un instant à partir duquel LEADER$_i$ contient toujours l'identité d'un processus correct. De plus, chaque processus correct envoie un message Phase1$(r, _, _)$.

Le prédicat de la ligne 05 est donc inéluctablement satisfait.

3. p_i est bloqué ligne 10. Observons que les deux points précedents montrent que tous les corrects terminent la première phase de la ronde r. Les arguments employés dans le point 1 montrent que p_i ne peut pas être bloqué indéfiniment à la ligne 10.

Nous obtenons une contradiction : le numéro de ronde r et le processus p_i sont choisis tels que p_i est bloqué lors d'une étape d'attente lorsqu'il exécute la ronde r. Or nous avons établi que p_i ne peut être bloqué lors des différentes étapes d'attente de la ronde r.

$\square_{Lemme\ 3.1}$

Pour un processus p_i qui termine l'exécution de la première phase de la ronde r, on note $aux_i[r]$ la valeur de la variable aux_i après sa mise à jour (ligne 08). Nous définissons également $AUX[r] = \{aux_i[r] \mid p_i$ termine la phase 1 de $r\ \}$.

Lemme 3.2 $\forall r : |\{v : v \in AUX[r] \wedge v \neq \perp\}| \leq z$.

Démonstration Soit p_i un processus qui termine la phase 1 de la ronde r. La variable aux_i est mise à jour avec une valeur $v \neq \perp$ lorsque p_i observe que les processus qui ont le même ensemble de leaders L forment une majorité (lignes 06-08). Dans ce cas, v est alors l'estimation d'un des processus qui appartiennent à L. Observons qu'il existe au plus un ensemble de leaders commun à une majorité de processus. Nous en déduisons que toutes les valeurs $aux_i \neq \perp$ à la fin de la phase 1 de la ronde r sont choisies parmi les estimations des processus qui appartiennent au même ensemble L. Comme cet ensemble est de taille au plus z, nous en concluons que $|\{aux_i[r] : aux_i[r] \neq \perp \wedge p_i$ termine la phase 1 de la ronde $r\}| \leq z$. $\square_{Lemme\ 3.2}$

Lemme 3.3 *(Aucun processus ne décide)* $\Rightarrow (\exists r : \perp \notin AUX[r])$.

Démonstration Par définition de la classe Ω^z, il existe un instant τ à partir duquel tous les processus ont de façon permanente le même ensemble de leaders L. Cet ensemble contient l'identité d'un processus correct (définition 3.3). Soit r une ronde qui démarre après τ. (i.e., le premier processus, par exemple p_i, qui exécute $r_i \leftarrow r$ le fait à un instant $\tau' > \tau$).

Comme nous supposons qu'aucun processus ne décide, tous les processus corrects exécutent la première phase de la ronde r. De plus, puisque cette phase démarre après l'instant τ, chaque processus correct diffuse un message Phase1$(r, L, _)$.

Par conséquent, tout processus p_i qui termine la phase 1 reçoit un message Phase1 d'un processus appartenant à $L = L_i$ et observe une majorité de processus avec le même ensemble L. Le prédicat des lignes 06-07 est satisfait et donc il n'existe pas de processus tel que $aux_i = \perp$ à la fin de la première phase de la ronde r. $\square_{Lemme\ 3.3}$

Proposition 3.3 *Pour $t < n/2$ et $z \leq k$, L'algorithme de la Figure 3.5 résout le (n, k)-accord dans le modèle $\mathcal{MP}_{n,t}[\Omega^z]$.*

Démonstration

Validité Remarquons que la valeur spéciale $\perp$ ne peut être décidée (lignes 12 et 13). De plus, les primitives de communications satisfont les propriétés d'intégrité et de validité : les variables est_i et aux_i contiennent donc toujours $\perp$ ou une valeur proposée (ligne Init).

Accord Il s'agit de montrer qu'au plus k valeurs sont décidées. Supposons qu'un processus décide. Soit r le plus petit numéro de ronde au cours de laquelle un processus décide (« décider au cours de la ronde r » est synonyme de « au cours de la ronde r, exécuter la ligne 13, la condition $\perp \notin rec_i$ étant satisfaite »). Nous montrons qu'il existe un ensemble de valeurs V de cardinal $\leq k$ tel que (1) toute valeur décidée au cours de la ronde r appartient à V et (2) toute valeur décidée au cours des rondes suivantes appartient également à v.

1. Soit $V = \{v : v \in AUX[r] \wedge v \neq \perp\}$. D'après le Lemme 3.2, $|V| \leq z$ d'où $|V[\leq k$ car $z \leq k$. Soit v une valeur décidée par un processus p_i au cours de la ronde r. L'ensemble $rec_i[r]$ des valeurs reçues par p_i au cours de la phase 2 de la ronde r (ligne 11) est un sous ensemble de $AUX[r]$, d'où $v \in AUX[r]$. Enfin, $v \neq \perp$ car p_i ne décide pas la valeur $\perp$ (ligne 13).

2. Supposons qu'un processus p_i décide une valeur $v \in V$ au cours de la ronde r. Nous montrons que les estimations $est_j[r]$ à la fin de la ronde r sont dans l'ensemble V.

   Par définition de l'ensemble V, les valeurs aux envoyées au cours de la phase 2 de la ronde r appartiennent à V. Soit p_j un processus qui termine la ronde r. Nous devons vérifier que p_j reçoit au cours la phase 2 un message Phase2$(r, v \neq \perp)$. La condition de la ligne 12 est alors satisfaite et, par suite, $est_j[r] \in V$.

   p_i décide au cours de la ronde r. Il a reçu $n - t > n/2$ messages Phase2 et chacun de ces messages transporte une valeur $aux \neq \perp$ (car $\perp \notin rec_i[r]$, ligne 13). Lors de la deuxième phase, p_j reçoit également $n - t > n/2$ messages Phase2. Il existe donc au moins un message Phase2 qui est reçu par p_i et p_j. Ce message transporte une valeur $aux \neq \perp$.

   Au début de la ronde $r+1$, toutes les estimations sont donc dans l'ensemble V. Par ailleurs, lors d'une ronde, lorsqu'un processus change son estimation, il choisit une nouvelle estimation parmi les estimations présentes au début de cette ronde. On en conclut que seules les valeurs $\in V$ peuvent être décidées au cours des rondes $\geq r$.

Terminaison Supposons qu'il n'existe pas de processus correct qui décide. Le Lemme 3.1 montre que les processus progressent de rondes en rondes indéfiniment. Par conséquent, il existe une ronde r tel que $\perp \notin AUX[r]$ (Lemme 3.2). Tous les messages Phase2(r, aux) sont donc tels que $aux \neq \perp$. Soit p_i un processus correct. p_i reçoit $n - t$ messages Phase2(r, aux) car tous les processus corrects effectuent la seconde phase de la ronde r et les messages sont transmis sans perte. Par suite, p_i décide (ligne 13 et tâche $T2$) : contradiction.

$\square_{Proposition\ 3.3}$

3.3.3 Optimalité

Nous établissons que dans la famille de modèles $(\mathcal{MP}_{n,t}[\Omega^z])_{0\leq t<n, 1\leq z\leq n}$, $t < n/2$ et $z \leq k$ sont des conditions nécessaires pour résoudre le (n,k)-accord. Ce résultat est obtenu par réduction à la question de la classe la plus faible dans la famille $(\diamond\mathcal{S}_x)_{1\leq x\leq n}$ qui permet de résoudre le (n,k)-accord qui a été résolu par Herlihy et Penso [71].

Lemme 3.4 *Il n'existe pas d'algorithme qui résout le (n,k)-accord dans le modèle $\mathcal{MP}_{n,t}[\Omega^z]$ si $(t \geq \frac{n}{2} \vee z > k)$.*

Démonstration Soit t, z tels que $t \geq n/2 \vee z > k$. Supposons qu'il existe un algorithme $\mathcal{A}$ pour le (n,k)-accord dans $\mathcal{MP}_{n,t}[\Omega^z]$. Nous montrerons dans le paragraphe 3.5 qu'il existe un algorithme $\mathcal{T}$ qui construit un détecteur de défaillances de la classe Ω^z dans le modèle $\mathcal{MP}_{n,t}[\diamond\mathcal{S}_{t-z+2}]$ (théorème 3.6). Un tel algorithme, indépendant de la valeur du paramètre t est présenté dans le paragraphe 3.4. En combinant $\mathcal{T}$ et $\mathcal{A}$, nous obtenons un algorithme qui résout le (n,k)-accord dans le modèle $\mathcal{MP}_{n,t}[\diamond\mathcal{S}_{t-z+2}]$. D'après la borne établie par Herlihy et Penso [71] sur la calculabilité du (n,k)-accord dans le modèle $\mathcal{MP}_{n,t}[\diamond\mathcal{S}_x]$, les paramètres t, z et k satisfont la relation $t < \min(n/2, (t-z+2)+k-1)$. Nous en concluons que $t < /2 \wedge z \leq k$: contradiction. $\square_{Lemme\ 3.4}$

Le théorème suivant caractérise la calculabilité du (n,k)-accord dans le modèle $\mathcal{MP}_{n,t}[\Omega^z]$.

Théorème 3.1 *Il existe un algorithme de (n,k)-accord dans le modèle $\mathcal{MP}_{n,t}[\Omega^z]$ si et seulement si $t < n/2 \wedge z \leq k$.*

Démonstration Ce théorème est un corollaire immédiat de la proposition 3.3 et du Lemme 3.4. $\square_{Théorème\ 3.1}$

3.4 Composition des classes $\diamond\mathcal{S}_x$ et $\diamond\psi^y$

Dans cette partie, nous étudions les relations entre les classes $\diamond\mathcal{S}_x$, $\diamond\psi^y$ et Ω^z. Nous présentons un algorithme qui construit un détecteur de défaillances de la classe Ω^z dans le modèle $\mathcal{MP}_{n,t}[\diamond\mathcal{S}_x, \diamond\psi^y]$. La construction requiert $x+y+z > t+1$. Nous montrons ensuite (paragraphe 3.5, théorème 3.3) que cette condition est nécessaire.

L'algorithme est divisé en deux composants appelés *roues* car ils « tournent » comme un engrenage jusqu'à ce qu'ils se synchronisent et stoppent leur mouvement. La rotation de la première roue (la roue « du bas ») dépend du comportement du détecteur $\diamond\mathcal{S}_x$ sous-jacent. Lorsqu'elle s'arrête, elle fournit une propriété qui permet à son tour à l'autre roue (la roue du « haut ») de se stabiliser. Comme nous allons le voir, la dénomination roue est liée à la structure des algorithmes. Chaque composant teste successivement une suite de configurations organisée logiquement en anneau jusqu'à identifier une configuration qui satisfait une certaine propriété.

3.4.1 Composant « roue du bas »

Ce composant maintient à jour sur chaque processus une variable REPR_i qui contient à tout instant une identité de processus. A l'instant τ, le processus p_j tel que $j = \text{REPR}_i^\tau$ est le *représentant* de p_i. L'algorithme essaie de fournir à un groupe X formé d'au moins x processus le même représentant, les processus à l'extérieur de ce groupe se considérant comme leur propre représentant. Plus précisément, le contenu des variables REPR_i doit satisfaire la propriété suivante :

Définition 3.8 (Spécification de $\diamond\mathcal{S}_x$, version représentant)
Représentant x-commun *Il existe un ensemble X de processus qui possède, à partir d'un certain temps toujours le même représentant. A partir de cet instant, le représentant d'un processus $\notin X$ est lui même.*
$\exists X \subseteq \Pi, \exists \ell \in Correct \cap X, \exists \tau$ *tels que*
(1) $(|X| \geq x) \wedge$
(2) $\forall \tau' \geq \tau : (i \in X \Rightarrow \text{REPR}_i^{\tau'} = \ell) \wedge$
(3) $\forall \tau' \geq \tau : (i \notin X \Rightarrow \text{REPR}_i^{\tau'} = i)$

Remarque Ce composant repose uniquement sur l'hypothèse que le détecteur sous-jacent appartient à la classe $\diamond\mathcal{S}_x$. D'autre part, il est clair qu'un détecteur qui fournit à chaque processus un représentant satisfaisant la propriété de *représentant x-commun* appartient à la classe $\diamond\mathcal{S}_x$. Par conséquent, la définition 3.8 est une spécification alternative de la classe $\diamond\mathcal{S}_x$. Dans le chapitre 4 , où nous cherchons à caractériser la puissance de calcul additionnelle offerte par un détecteur, nous utiliserons cette définition alternative.

Fonctionnement de l'algorithme Sur chaque processus p_i, la mise à jour de la variable REPR_i est fondée sur le contenu courant de la variable TRUSTED_i contrôlée par un détecteur $\diamond\mathcal{S}_x$. Soit $\mathcal{X}$ l'ensemble de tous les sous-ensembles de $\Pi = \{p_1, \ldots, p_n\}$ de cardinal x. Soit $nb\_x = \binom{n}{x}$ le nombre de ces sous-ensembles ($|\mathcal{X}| = nb\_x$). Nous ordonnons arbitrairement les éléments de $\mathcal{X}$ et pour k, $1 \leq k \leq nb_x$ nous notons $\mathcal{X}[k]$ le $k$ième élément de $\mathcal{X}$ dans l'ordre choisi. De même, pour chaque ensemble $\mathcal{X}[k]$, nous rangeons les identités qui le composent selon un ordre arbitrairement choisi : $\ell_k^1, \ldots, \ell_k^x$. Ainsi, la suite infinie $\mathcal{X}[1]$, $\mathcal{X}[2]$,..., $\mathcal{X}[nb\_x]$,$\mathcal{X}[1]$, $\mathcal{X}[2]$,..., $\mathcal{X}[nb\_x]$,$\mathcal{X}[1]$,... donne naissance à la suite infinie d'identités de processus $\ell_1^1, \ldots, \ell_x^1, \ell_1^2, \ldots, \ell_x^2, \ell_1^3, \ldots$ (voir Figure 3.6). Cette suite est supposée connue initialement par tous les processus pour qu'ils puissent la parcourir dans le même ordre.

En plus de la variable REPR_i, chaque processus p_i dispose de deux variables locales X_i et ℓ_i initialisées respectivement à $\mathcal{X}[1]$ et ℓ_1^1 (la première identité de l'ensemble $\mathcal{X}[1]$). Pour progresser dans la suite infinie d'identités, il utilise la fonction $\mathsf{Next}(\_, \_)$. $\mathsf{Next}(\ell_y^k, \mathcal{X}[k])$ renvoie le couple $(\ell_{y+1}^k, \mathcal{X}[k])$ si $y < x$ et le couple $(\ell_1^{k+1}, \mathcal{X}[k+1])$ si $y = x$ ($k+1$ étant remplacé par 1 lorsque $k = nb\_x$).

Le comportement du composant roue du bas pour le processus p_i est décrit dans la Figure 3.7. Les processus testent successivement les ensembles de la suite infinie

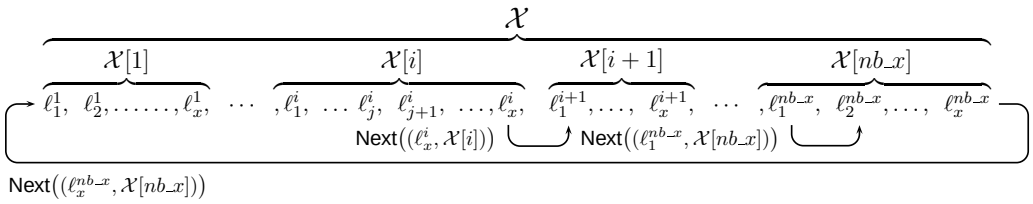

FIG. 3.6 – La fonction Next() sur l'anneau logique (ℓ, X)

init : $X_i \leftarrow \mathcal{X}[1]$; $\ell x_i \leftarrow \ell_1^1$; $\text{REPR}_i \leftarrow i$

Task $T1$:
  repeat forever
    if $(i \in X_i)$ **then** $\text{REPR}_i \leftarrow \ell x_i$ **else** $\text{REPR}_i \leftarrow i$ **endif** ;
    if $\big((i \in X_i) \wedge (\ell x_i \notin \text{TRUSTED}_i)\big)$ **then** *R_Broadcast* X_move$(\ell x_i, X_i)$ **endif**
  endrepeat

Task $T2$: **when** X_move$(\ell x_i, X_i)$ **is R_delivered** : $(\ell x_i, X_i) \leftarrow$ Next$(\ell x_i, X_i)$

FIG. 3.7 – De $\diamond\psi^y + \diamond\mathcal{S}_x$ vers Ω^z : roue du bas (code pour p_i)

$\mathcal{X}[1], \ldots, \mathcal{X}[nb\_x]$, $\mathcal{X}[1], \ldots$ jusqu'à ce qu'il se stabilisent sur l'un d'entre eux. X_i représente les processus chargés d'extraire un représentant commun ℓx_i. Pour cela, chaque processus p_i dont l'identité apparaît dans X_i utilise l'ensemble TRUSTED_i contrôlé par le détecteur sous-jacent $\diamond\mathcal{S}_x$. Si tous les processus de X_i font confiance à l'un d'entre eux (ℓx_i contient alors l'identité de ce processus), ils n'envoient plus de messages X_move(). Par contre, si un processus p_j de X_i ne fait pas confiance à son représentant courant ℓx_j, il envoie par l'intermédiaire d'une primitive de diffusion fiable un message X_move$(\ell x_j, X_i)$ pour indiquer que, de son point de vue, ℓx_j ne peut être le représentant commun de l'ensemble X_i. Dans la tâche $T2$, le traitement d'un message X_move(X, ℓ) par le processus p_i n'a lieu que lorsque $X_i = X$ et $\ell x_i = \ell$. Il teste alors la prochaine identité (selon l'ordre de la séquence infinie) et éventuellement le prochain ensemble $\mathcal{X}[k+1]$ candidat si $X_i = \mathcal{X}[k] = X$ et $\ell x_i = \ell_k^x = \ell$ est la dernière identité de $\mathcal{X}[k]$.

Enfin, considérons le cas où les processus progressent jusqu'à rencontrer un ensemble X qui contient uniquement des processus défaillants. On vérifie aisément que les processus p_i non-défaillants (1) bouclent indéfiniment dans la tâche $T1$ sans jamais envoyé de message X_move() et (2) sont alors en permanence tels que $\text{REPR}_i = i$.

Preuve de l'algorithme La démonstration considère une exécution infinie arbitraire. Nous ne faisons pas d'hypothèse particulière sur t $(1 \leq t < n)$.

Lemme 3.5 *$\forall i \in Correct$, il existe une paire (λ_i, σ_i) et un instant τ_i tels que $\forall \tau \geq \tau_i$:* $(\ell x_i^\tau, X_i^\tau) = (\lambda_i, \sigma_i)$.

Démonstration La preuve du lemme repose sur le fait suivant :

Fait 3.1 *Il existe une paire (ℓ, X) telle que le nombre de messages X_ move$\big(\ell, X\big)$ envoyés est fini.*

Démonstration La preuve est une analyse de cas. Nous considérons deux cas, en fonction du nombre de pannes effectives f.

1. $f \geq x$. Soit X un ensemble de x processus fautifs. Soit ℓ l'identité d'un processus arbitrairement choisi dans X. D'après le texte de l'algorithme, seuls les processus de l'ensemble X peuvent envoyer des messages X_move$\big(\ell, X\big)$. Puisque tous ces processus sont défaillants dans l'exécution considérée, le nombre de messages X_move$\big(\ell, X\big)$ envoyés est fini.
2. $f < x$. La propriété de précision faible du détecteur $\diamond\mathcal{S}_x$ garantit l'existence d'un ensemble $X \subseteq \Pi$ de cardinal au moins x ainsi que l'existence d'un processus correct $p_\ell, \ell \in X$ tels que, à partir d'un certain instant τ, p_ℓ n'est pas suspecté par les processus appartenant à X. De façon plus formel, on a $\forall\tau' \geq \tau, \forall i \in X : \ell \in \text{TRUSTED}_i^{\tau'}$. Examinons maintenant le code : (1) seuls les processus dont l'identité est dans l'ensemble X peuvent envoyer des messages X_move$\big(\ell, X\big)$. De plus (2), un processus p_i tel que $i \in X$ ne peut diffuser (appel *R_broadacast*) le message X_move$\big(\ell, X\big)$ qu'à la condition que $\ell \notin \text{TRUSTED}_i$. On en déduit qu'à partir de l'instant τ, aucun nouveau message X_move$\big(\ell, X\big)$ n'est diffusé, ce qui implique que le nombre de tels messages générés est fini.

$\square_{fait\ 3.1}$

Soit p_i un processus correct. Supposons qu'il n'existe pas de paire (λ_i, σ_i) telle que, après un certain instant le prédicat $(\ell x_i, X_i) = (\lambda_i, \sigma_i)$ soit toujours vérifié. Les paires $(\ell x, X)$ sont logiquement organisées en anneau (voir Figure 3.6). Lorsque la valeur de $(\ell x_i, X_i)$ change, elle est remplacée par la paire qui lui succède dans l'anneau logique. Les valeurs successives de $(\ell x_i, X_i)$ sont donc $(\ell_1^1, \mathcal{X}[1]), (\ell_1^2, \mathcal{X}[1]), \ldots, (\ell_x^{nb\_x}, \mathcal{X}[nb\_x]), (\ell_1^1, \mathcal{X}[1])$, etc. Par conséquent, $(\ell x_i, X_i)$ contient chacune des valeurs $(\ell_\alpha^\beta, \mathcal{X}[\beta]), 1 \leq \alpha \leq x, 1 \leq \beta \leq nb\_x$ infiniment souvent. En particulier, p_i exécute $(\ell x_i, X_i) \leftarrow \mathsf{Next}(\ell, X)$ infiniment souvent. Ceci contredit le fait 3.1 car l'exécution de cette instruction est déclenchée par la réception du message X_move(ℓ, X). Or, dans l'exécution considérée, un nombre fini de ces messages est diffusé. Il existe donc une paire (λ_i, σ_i) et un instant τ_i tels que $\forall\tau \geq \tau_i : (\ell x_i^\tau, X_i^\tau) = (\lambda_i, \sigma_i)$. $\square_{Lemme\ 3.5}$

Corollaire 3.1 *Il existe un instant après lequel l'algorithme est quiet (i.e., dans toute exécution, il existe un instant à partir duquel les processus ne diffuse plus de messages X_ move).*

Démonstration Supposons qu'il existe un processus correct p_i qui diffuse toujours de nouveaux messages X_move. Nous établissons une contradiction. Il existe un instant τ_i à partir duquel $(\ell x_i, X_i)$ contient toujours la constante (λ_i, σ_i) (lemma 3.5). Donc tous les messages diffusés par p_i après τ_i transportent (λ_i, σ_i). Puisque la diffusion est fiable (propriétés de validité et de terminaison, cf. définition 1.1) et p_i diffuse une infinité de message X_move(), il existe un instant $\tau > \tau_i$ auquel p_i reçoit un message

X_move(λ_i, σ_i). D'après le texte de l'algorithme, il change alors la valeur de $(\ell x_i, X_i)$ en exécutant $(\ell x_i, X_i) \leftarrow \mathsf{Next}(\lambda_i, \sigma_i)$: contradicition avec le Lemme 3.5. $\square_{Corollaire\ 3.1}$

Lemme 3.6 $\forall i, j \in Correct : (\lambda_i, \sigma_i) = (\lambda_j, \sigma_j)$. *(Dans la suite, cette paire est notée* (λ, σ)*.)*

Démonstration Soient deux processus corrects p_i et p_j. Puisque les messages X_move sont diffusés de façon fiable, p_i et p_j reçoivent le même (multi)-ensemble de messages. Ce multi-ensemble est fini (corollaire 3.1). Enfin, ces messages sont consommés par p_i et p_j dans le même ordre défini par l'organisation en anneau des paires (ℓ, X). La consommation du message X_move(ℓ, X) par p_i (ou p_j) entraîne la mise à jour de (ℓ, X_i) (respectivement (ℓ_j, X_j)) avec la valeur $\mathsf{Next}(\ell, X)$ (tâche $T2$). On en déduit que $(\lambda_i, \sigma_i) = (\lambda_j, \sigma_j)$. $\square_{Lemme\ 3.6}$

Lemme 3.7 $(\sigma \cap Correct \neq \emptyset) \Rightarrow (\lambda \in Correct)$.

Démonstration Supposons que $\sigma \cap Correct \neq \emptyset$ et λ est l'identité d'un processus fautif. Il existe un process p_i tel que $i \in \sigma \cap Correct$. D'après la propriété de complétude forte du détecteur $\diamond\mathcal{S}_x$ (définition 3.1), il existe un instant τ à partir duquel le prédicat $\lambda \notin \text{TRUSTED}_i$ est toujours satisfait. De plus, il existe un instant τ_i à partir duquel on a toujours $(\ell x_i, X_i) = (\lambda, \sigma)$ (Lemmes 3.5 et 3.6). D'après le texte de l'algorithme, il existe donc un instant $\tau' > \max(\tau_i, \tau)$ auquel p_i diffuse un message X_move(λ, σ). Puisque que la diffusion est fiable, ce message est *R_livré* par p_i. Cet evénement déclenche l'exécution de $(\ell x_i, X_i) \leftarrow \mathsf{Next}(\lambda, \sigma)$. Il existe donc un instant $> \tau_i$ tel que, à cet instant $(\ell x_i, X_i) \neq (\lambda, \sigma)$: une contradiction avec le Lemme 3.5. $\square_{Lemme\ 3.7}$

Proposition 3.4 *L'algorithme décrit à la Figure 3.7 garantit l'existence d'un ensemble* $X \subseteq \Pi$*, d'une identité* $\rho \in \Pi$ *et d'un instant* τ *tels que,* $\forall \tau' \geq \tau$*, les propriétées suivantes sont satisfaites :*

1. $|X| = x$;
2. $i \in (\Pi - X) \cap Correct \Rightarrow repr_i = i$;
3. $\forall i, j \in X \cap Correct : repr_i = repr_j = \rho$;
4. $\rho \in Correct \cap X$.

Démonstration Définissons $\tau = \max\{\tau_i : i \in \Pi\}$, où τ_i est l'instant introduit dans le Lemme 3.5 et $(\rho, X) = (\lambda, \sigma)$, où (λ, σ) est le couple introduit dans le Lemme 3.6. Nous vérifions que le triplet τ, ρ, X vérifie les 4 propriétés énoncées ci-dessus.

1. Par définition, σ est un ensemble de la suite $\mathcal{X}$. D'où $|X| = |\sigma| = x$.
2. Soit $i \in \Pi - X$ et supposons que p_i est correct. Après l'instant τ, on a toujours $X_i = X$ (Lemme 3.5). D'après le texte de l'algorithme, $\forall \tau' \geq \tau : \text{REPR}_i^{\tau'} = i$.
3. Soient $i, j \in X \cap Correct$. Après l'instant τ le prédicat suivant est toujours vrai : $X_i = X_j = X$ (choix de X ; Lemmes 3.5 et 3.6). D'après le texte de l'algorithme, $\forall \tau', \tau'' \geq \tau : \text{REPR}_i^{\tau'} = \text{REPR}_j^{\tau''} = \lambda = \rho$.

4. $\rho = \lambda \in \mathit{Correct}$ (Lemme 3.7). De plus $\rho \in X$ par définition de la paire (λ, σ).

$\square_{\mathit{Proposition}\ 3.4}$

3.4.2 Roue du haut

Principes et description Le composant « roue du haut » (Figure 3.8) se décompose en 4 tâches exécutées en parallèle. L'algorithme fonctionne selon le même principe que le composant « roue du bas ». Il utilise une suite notée $\mathcal{L}$ qui contient tous les sous-ensembles de Π de cardinal $z = (t+2) - (x+y)$. La suite $\mathcal{L}$ est supposée connue initialement de tous les processus. Nous notons $nb\_L = \binom{n}{(t+2)-(x+y)}$ le nombre d'éléments dans la suite $\mathcal{L}$ et $\mathcal{L}[k]$ son $k$ième élément. La fonction $\mathsf{Next}(\mathcal{L}[k])$ renvoie $\mathcal{L}[k+1]$ si $k < nb\_L$ et $\mathcal{L}[1]$ si $k = nb\_L$.

```
Init : L_i ← L[1]

Task T3 :
(1) repeat forever
(2)     Broadcast Inquiry();
(3)     wait until ( corresponding Response(id) received from ≥ n − NB_C_i processes )
                                                  % NB_C_i can dynamically change
(4)     let rec_from_i = {id_j received previously at line 3 };
(5)     if (rec_from_i ∩ L_i = ∅) then R_Broadcast L_move(L_i) endif
(6) enddo

Task T4 : when L_move(L_i) is R_delivered : L_i ← Next(L_i)

Task T5 : when Inquiry() is received from p_j : send Response(REPR_i) to p_j

Task T6 : when LEADER_i is read by the upper layer : return(L_i)
```

FIG. 3.8 – De $\diamond\psi^y + \diamond\mathcal{S}_x$ vers Ω^z : roue du haut (code pour p_i)

Le but de cet algorithme est d'implémenter un détecteur Ω^z avec $z = (t+2) - (x+y)$. Pour cela, chaque processus p_i bénéficie de (1) la sortie du composant roue du bas (variable REPR_i dont les valeurs successives respectent la définition 3.8) et (2) l'information fournie par le détecteur $\diamond\psi^y$ sous-jacent (variable NB_C_i, définition 3.7). La variable L_i, régulièrement mise à jour par l'algorithme émule la sortie d'un détecteur Ω^z (tâche $T6$). Elle contient à chaque instant un ensemble de z identités et l'on souhaite qu'à partir d'un certain temps, cet ensemble ne change plus et inclue l'identité d'un processus correct.

Les processus parcourent dans le même ordre la suite infinie $\mathcal{L}[1], \mathcal{L}[2], \ldots, \mathcal{L}[nb\_L], \mathcal{L}[1], \ldots$. La variable locale L_i, initialisée à $\mathcal{L}[1]$ contient à chaque instant l'ensemble en cours d'examen par p_i. L'examen d'un ensemble L_i par p_i se déroule ainsi :

1. p_i essaie de déterminer si l'ensemble L_i contient l'identité d'un processus correct. Pour cela, il diffuse régulièrement des messages Inquiry (ligne 2, tâche $T3$). Lorsque

qu'un processus p_j reçoit un tel message, il répond par un message Response qui transporte l'identité de son représentant courant (tâche $T5$).

2. p_i passe ensuite en mode attente. Cette attente se termine lorsque p_i a reçu des messages Response en provenance de $n - \text{NB\_C}_i$ processus. Par définition de $\diamond\psi^y$ (def. 3.7), nous savons qu'à partir d'un certain temps, nous avons toujours $n - \text{NB\_C}_i = n - \max(t-y, f) \leq n-f$. Or p_i reçoit au moins un message Response de chacun des $(n-f)$ processus corrects. Cette phase d'attente n'est donc pas bloquante.

3. Enfin, après avoir reçu suffisamment de messages Response, p_i collecte les identités id_j transportées par ces messages dans l'ensemble rec_i (ligne 3). Si l'une de ces identités est contenue dans L_i, p_i considère que L_i a passé l'examen. En effet, le composant roue du bas assure qu'à partir d'un certain chaque processus représentant est correct. Dans le cas contraire, du point de vue de p_i tous les processus de L_i sont défaillants. Il diffuse donc de façon *fiable* un message L_move(L_i) pour informer les autres processus (ligne 5) qu'ils doivent tester l'ensemble qui suit L_i dans l'anneau $\mathcal{L}$.

L'idée d'un engrenage entre les deux composants se retrouve par exemple dans le scénario suivant. Supposons que les deux roues cessent de tourner, c'est à dire que le contenu des variable L_i et REPR_i soit stable. Plus tard, le représentant x-commun tombe en panne. La roue du bas se remet alors à tourner et son mouvement entraîne éventuellement le redémarrage de la roue du haut. Le démarrage de la roue du haut dépend de la combinaison de plusieurs paramètres tels que les délais de transmissions des messages Response, l'ensemble L sur lequel la roue du haut s'est arrêtée et le nombre de couples (X, ℓ) à tester par les processus avant d'identifier une configuration stable. Néanmoins, il existe des cas où la rotation de la roue du haut est directement liée au mouvement de l'autre roue.

Démonstration de l'algorithme Les deux algorithmes sont fondés sur le même principe : les processus parcourent l'ensemble des configurations possibles qui sont organisées en anneau jusqu'à identifier une configuration qui satisfait une propriété donnée. Lorsqu'une configuration ne satisfait pas la propriété recherchée (ici une configuration est un ensemble L de processus de cardinal z ; la propriété recherchée l'existence d'un processus correct dans cet ensemble), un processus correct le détecte et diffuse de façon fiable cette information (messages *_Move(Config.)). La réception d'un tel message entraîne l'examen de la configuration suivante dans l'anneau. La démonstration de la correction de l'algorithme est donc largement inspirée de la démonstration précédente.

Comme d'habitude, nous considérons un exécution infinie arbitraire. *Correct* désigne l'ensemble des identités des processus corrects dans cette exécution et var_i^τ dénote la valeur de la variable locale var_i du processus p_i à l'instant τ.

Lemme 3.8 *$\forall i \in Correct$, il existe un ensemble Λ_i et un instant τ_i tels que $\forall \tau \geq \tau_i$:* $L_i^\tau = \Lambda_i$.

Démonstration La démonstration repose sur le fait suivant, qui est démontré un peu plus loin :

Fait 3.2 *Il existe un ensemble d'identités L tel que le nombre de messages L_move(L) diffusés est fini.*

Soit p_i un processus correct. Supposons par contradiction qu'il n'existe pas d'ensemble Λ_i tel que, après un certain instant, $L_i = \Lambda_i$ est toujours vrai. Les ensembles L sont logiquement organisés en anneau. Lorsque la valeur de L_i change, elle est remplacée par l'ensemble qui lui succède dans l'anneau logique. Les valeurs successives de L_i sont donc $\mathcal{L}[1], \mathcal{L}[2], \ldots, \mathcal{L}[nb\_L], \mathcal{L}[1], \ldots$. Par conséquent, L_i contient chacun des ensembles $\mathcal{L}[\beta], 1 \leq \beta \leq nb\_L$ infiniment souvent. En particulier, p_i exécute $L_i \leftarrow \mathsf{Next}(L)$ infiniment souvent. Ceci contredit le fait 3.2 car l'exécution de cette instruction est déclenchée par la réception du message L_move(L). Or, dans l'exécution considérée, un nombre fini de ces messages est diffusé. Il existe donc un ensemble Λ_i et un instant τ_i tels que $\forall \tau \geq \tau_i : L_i^\tau = \Lambda_i$.

Fait 3.2 Il existe un ensemble d'identités L tel que le nombre de messages L_move(L) diffusés est fini.

Démonstration La roue du bas effectue un nombre fini de tours. Nous considérons un instant τ_0 à partir duquel la roue du bas ne bouge plus. A partir de cet instant, il existe un ensemble $X \subseteq \Pi, |X| = x$ et une identité λ tels que (proposition 3.4) :

1. $\forall i \in \Pi - X, \forall \tau \geq \tau_0 : repr_i^\tau = i$ et,
2. (a) $X \cap Correct \neq \emptyset \Rightarrow (\lambda \in Correct \cap X) \wedge (\forall i \in X, \forall \tau \geq \tau_0 : repr_i^\tau = \lambda)$ ou,
   (b) $X \cap Correct = \emptyset \Rightarrow$ tous les processus dont l'identité est dans l'ensemble X sont tombés en panne avant τ_0.

Choisissons un ensemble $L \subseteq \Pi$ qui satisfait les conditions suivantes (voir aussi la Figure 3.9).

- $|L| = (t+2) - (x+y)$;
- $|X \cap L| = 1$;
- $(X \cap Correct \neq \emptyset) \Rightarrow X \cap L = \{\lambda\}$;
- $L \cap Correct \neq \emptyset$.

Observons que, dans toute exécution infinie, il est possible de choisir un tel ensemble. De plus il y a dans L au moins une identité ℓ tel que p_ℓ est un processus correct qui est, au bout d'un certain temps, toujours son propre représentant ($\text{REPR}_\ell = \ell$).

Pour montrer qu'un nombre fini de messages L_move(L) est diffusé, nous examinons deux cas selon le nombre de processus fautifs f.

- $f \geq t - y + 1$.
  Il existe un instant τ_1 à partir duquel la valeur fournie par le détecteur $\diamond\psi^y$ est toujours f. Plus précisément : $\forall \tau \geq \tau_1, \forall i \in Correct$: $\text{NB\_C}_i^\tau = f$ (propriété de convergence inéluctable de la classe $\diamond\psi^y$, déf. 3.7). Soit τ_2 un instant après lequel aucune nouvelle défaillance ne se produit. De plus, on suppose que chaque

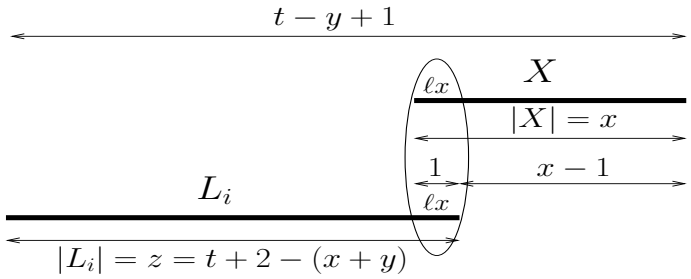

FIG. 3.9 – Arrêt de la roue du haut

message Inquiry ou Response envoyé par un processus fautif à destination d'un processus correct a été reçu et traité par son destinataire avant τ_2.
Soit $\tau_s = \max(\tau_0, \tau_1, \tau_2)$. Après τ_s, l'état du système est stable : aucun processus ne tombe en panne, les sorties du détecteur $\diamond\psi^y$ et de la roue du bas ne changent plus. Soit $i \in Correct$. Après τ_s, lors de chaque exécution de la boucle « **repeat** », p_i reçoit exactement un message Response de chaque processus correct.
Par choix de L, il existe un processus correct p_ℓ, $\ell \in L$ tel que $\text{REPR}_\ell = \ell$ est toujours vrai à partir de τ_0. Donc, à partir d'un certain instant, tous les messages Response(id_ℓ) envoyés par p_ℓ sont tels que $id_\ell = \ell$. Puisque que p_i reçoit toujours après τ_s un message de p_ℓ, on a toujours à partir d'un certain temps $\ell \in rec\_from_i$. Donc, le prédicat $rec\_from_i \cap L$ est, à partir d'un certain temps, toujours vrai. Par conséquent, à partir de cet instant p_i ne diffuse plus de message L_move(L) (ligne 5). Nous en déduisons que le nombre de messages L_move(L) générés dans l'exécution considérée est fini.

– $f < t - y + 1$.
Dans ce cas, il existe un instant τ_1 tel que $\forall \tau \geq \tau_1, \forall i \in Correct : \text{NB\_C}_i^\tau = t - y$. De la même manière que dans l'analyse du cas précédent, on considère l'instant τ_2 après lequel il n'y a plus de défaillances ni de réception de message de processus fautifs par les processus corrects, ainsi que l'instant $\tau_s = \max(\tau_0, \tau_1, \tau_2)$.
Soit $i \in Correct$. Nous montrons d'abord qu'à partir de l'instant τ_s, p_i reçoit toujours au moins un message Response() d'un processus p_j tel que $j \in L \cup X$. p_i attend la réception de messages Response() en provenance de $n - \text{NB\_C}_i = n - (t - y)$ processus distincts. Par ailleurs, par choix de L, nous avons $|L \cup X| = |L| + |X| - |L \cap X| = (t + 2) - (x + y) + x - 1 = 1 + (t - y)$. Par conséquent, p_i reçoit et traite un message d'un processus p_j tel que $j \in L \cup X$.
La sortie du composant roue du bas se stabilise (à l'instant τ_0). A partir d'un certain instant, les messages Response(id_j) envoyés par un processus correct p_j, $j \in L \cup X$ sont donc toujours tels que $id_j = j$ si $j \in L - X$ ou $id_j = \lambda$ si $j \in X$. Dans les deux cas, $id_j \in L$. A chaque tour de la boucle « **repeat** », rec_from_i est mis à jour avec les identités id recoltées lors de la réception des messages Response(). On en déduit qu'il existe un instant à partir duquel le prédicat $rec\_from_i \cap L \neq \emptyset$ est toujours vrai. Le nombre de message L_move(L) diffusé par chaque processus correct p_i est donc fini. Comme un processus fautif ne diffuse qu'un nombre fini

de messages, on en déduit que le nombre de messages L_move(L) générés est fini.
$\square_{fait\ 3.2}$

$\square_{Lemme\ 3.8}$

Corollaire 3.2 *Il existe un instant à partir duquel les processus ne diffusent plus de messages L_move().*

Démonstration Supposons qu'il existe un processus correct p_i qui diffuse toujours de nouveaux messages L_move. Nous établissons une contradiction. Il existe un instant τ_i à partir duquel L_i contient toujours l'ensemble Λ_i (lemma 3.8). Après τ_i, p_i diffuse donc uniquement des messages L_move(Λ_i). Par conséquent, puisque la diffusion est fiable, il existe un instant $\tau > \tau_i$ auquel p_i reçoit un message L_move(Λ_i). D'après le texte de l'algorithme, il change alors la valeur de L_i en exécutant $L_i \leftarrow \mathsf{Next}(\Lambda_i)$: contradiction avec le Lemme 3.8. $\square_{Corollaire\ 3.2}$

Lemme 3.9 $\forall i, j \in Correct : \Lambda_i = \Lambda_j$. *(Dans la suite, cet ensemble est noté Λ.)*

Démonstration Soient deux processus corrects p_i et p_j. Puisque les messages L_move sont diffusés de façon fiable, p_i et p_j reçoivent le même multi-ensemble de messages (propriétés de la diffusion fiable, def. 1.1). Ce multi-ensemble est fini (corollaire 3.2). Enfin, ces messages sont consommés par p_i et p_j dans le même ordre défini par l'organisation en anneau des ensembles L. La consommation du message L_move(L) par p_i (ou p_j) entraîne la mise à jour de L_i (respectivement L_j) avec la valeur $\mathsf{Next}(L)$ (tâche $T4$). On en déduit que $\Lambda_i = \Lambda_j$. $\square_{Lemme\ 3.9}$

Proposition 3.5 *Soit x, y et t tels que $1 \leq x \leq t+1$, $0 \leq y \leq t$ et $x + y < t + 2$. Dans le modèle $\mathcal{MP}_{n,t}[\diamond\mathcal{S}_x, \diamond\psi^y]$, l'algorithme de la Figure 3.8 simule un détecteur de défaillances de la classe Ω^z tel que $z = (t+2) - (x+y)$.*

Démonstration La sortie de l'algorithme est l'ensemble des valeurs des variables $L_i, i \in \Pi$ (tâche $T6$). Nous établissons que ces valeurs satisfont la propriété leaders multiples (définition 3.3). Il existe un ensemble Λ et un instant τ tels que $|\Lambda| = z$ et $\forall i \in Correct, \forall \tau' \geq \tau : L_i^{\tau'} = \Lambda$ (Lemmes 3.8 et 3.9). Il reste à montrer que $\Lambda \cap Correct \neq \emptyset$.

Supposons que $\Lambda \cap Correct = \emptyset$. Soit p_i un processus correct. D'après les propriétés assurées par la roue du bas (proposition 3.4), il existe un instant à partir duquel $\forall j \in Correct, \text{REPR}_j \in Correct$. Donc, d'après le texte du protocole, il existe un instant τ_1 à partir duquel le prédicat $rec\_from_i \subseteq Correct$ est toujours vrai. Par ailleurs, à partir de l'instant τ on a toujours $L_i = \Lambda \subseteq \Pi - Correct$. Par conséquent, à un certain instant $\tau_2 > \max(\tau, \tau_1)$, le prédicat local $L_i \cap rec\_from_i \neq \emptyset$ n'est pas satisfait. D'après le texte de l'algorithme, un message L_move(Λ) est alors diffusé par p_i. Puisque la diffusion est fiable, p_i reçoit et traite ce message à un instant $> \tau_2 > \tau$. A cet instant, $L_i = \Lambda$ (Lemmes 3.8 et 3.9), et p_i exécute alors $L_i \leftarrow \mathsf{Next}(\Lambda)$: contradiction avec le Lemme 3.9. $\square_{Proposition\ 3.5}$

3.5 Optimalité et impossibilités des compositions

Dans le paragraphe précédent, nous avons montré en exhibant un algorithme qu'il est possible de composer deux détecteurs $\diamond\psi^y$ et $\diamond\mathcal{S}_x$ pour obtenir un détecteur Ω^z à condition que $x+y+z > t+1$. Le résultat principal établi dans ce qui suit est l'optimalité de cette borne (théorème 3.3). Nous donnons ensuite la preuve des réductions ou de leur impossibilité résumées dans la Figure 3.1.

Préliminaires Essentiellement, les démonstrations établissent les résultats par réduction. Nous démontrons l'optimalité ou l'impossibilité des compositions par réductions au problème de calculabilité du (n,k)-accord dans le modèle $\mathcal{MP}_{n,t}[\mathcal{S}_x]$. Dans ce modèles, des conditions nécessaires et suffisantes sur les paramètres k, t et x ont été établies. Plus précisèment, Herlihy et Penso ont démontré le thèorème suivant :

Théorème 3.2 ([71]) *Il existe un solution au problème du (n,k)-accord dans le modèle $\mathcal{MP}_{n,t}[\mathcal{S}_x]$ si et seulement si $t < k + x - 1$.*

Il est clair qu'un détecteur $\mathcal{S}_1$ n'apporte aucune information additionnelle sur les défaillance. Il existe une implémentation triviale d'un détecteur de cette classe dans le modèle $\mathcal{MP}_{n,t}[\emptyset]$: sur chaque processus p_i, $\text{TRUSTED}_i = \{i\}$. Ainsi, si un problème a une solution dans le modèle $\mathcal{MP}_{n,t}[\mathcal{S}_1]$ alors ce problème possède une solution dans le modèle $\mathcal{MP}_{n,t}[\emptyset]$. On en déduit le corollaire suivant :

Corollaire 3.3 *Il existe une solution au problème du k-accord dans le modèle $\mathcal{MP}_{n,t}[\emptyset]$ si et seulement si $t < k$.*

Par ailleurs, nous remarquons qu'un détecteur $\diamond\psi^y$ (ou ψ^y) n'apporte pas de puissance d'additionnelle dans les exécutions où le nombre de défaillances f est inférieur à $t - y$. En effet, dans toute exécution qui comporte au plus $t - y$ défaillances, la sortie d'un détecteur $\diamond\psi^y$ (ou ψ^y) est au bout d'un certain temps toujours $t - y$. La seule information dont bénéficient les processus est que le nombre de défaillances est au plus $t - y$. Ils sont incapables de discerner à partir des seules informations fournies par le détecteur une exécution dans laquelle tous les processus sont corrects d'une exécution dans laquelle $t-y$ processus tombent en panne. Cette observation est utilisée à plusieurs reprises dans les démonstrations de cette partie.

Enfin, comme nous avons montré dans le paragraphe 3.2 que les classes $\diamond\psi^y/\psi^y$ et $\diamond\phi^y/\phi^y$ sont équivalentes, nous utilisons parfois $\diamond\phi^y/\phi^y$ au lieu de $\diamond\psi^y/\psi^y$.

Optimalité des deux roues : borne pour l'addition des classes $\diamond\mathcal{S}_x$ et $\diamond\psi^y$

Théorème 3.3 *Il existe un algorithme qui implémente un détecteur Ω^z dans le modèle $\mathcal{MP}_{n,t}[\diamond\mathcal{S}_x, \diamond\psi^y]$ si et seulement si $x + y + z > t + 1$.*

Démonstration [direction $\Leftarrow$] Il existe un algorithme qui implémente un détecteur de défaillances de la classes Ω^z dans le modèle $\mathcal{MP}_{n,t}[\diamond\mathcal{S}_x, \diamond\psi^y]$: l'algorithme « des deux roues » présenté dans le paragraphe 3.4. La démonstration de sa correction suppose

$x + y + z > t + 1$ (proposition 3.5).

[direction $\Rightarrow$] Nous établissons que $x + y + z > t + 1$ est une condition nécessaire par contradiction. Supposons qu'il existe un algorithme $\mathcal{T}$ qui implémente un détecteur de la classe Ω^z dans le modèle $\mathcal{MP}_{n,t}[\mathcal{S}_x, \psi^y]$ avec $x + y + z \leq t + 1$. Un résultat d'impossibilité établi dans le modèle $\mathcal{MP}_{n,t}[\mathcal{S}_x, \psi^y]$ s'étend immédiatement au modèle $\mathcal{MP}_{n,t}[\diamond\mathcal{S}_x, \diamond\psi^y]$ car $\mathcal{S}_x \subseteq \diamond\mathcal{S}_x$ et $\psi^y \subseteq \psi^y$.

Considérons l'ensemble $\mathcal{E}$ des exécutions du modèle $\mathcal{MP}_{n,t}[\mathcal{S}_x, \psi^y]$ dans lesquelles le nombre de défaillances est majorée par $t - y$. Dans chacune de ces exécutions, il existe un instant (qui dépend de l'exécution considérée) à partir duquel on a toujours $\text{NB\_C}_i = t - y$ (« vivacité » de la classe ψ^y). Donc, si on fait l'hypothèse que le nombre de défaillances est borné par $t - y$, il existe une implémentation triviale d'un détecteur de la classe ψ^y : pour chaque processus p_i, on attribue de façon permanente la valeur $t - y$ à la variable NB_C_i.

Par conséquent, en utilisant la transformation $\mathcal{T}$, il est possible de construire un détecteur de la classe Ω^z dans le modèle $\mathcal{MP}_{n,t-y}[\mathcal{S}_x]$. De plus, on sait que l'on peut résoudre le problème (n, z)-accord dans le modèle $\mathcal{MP}_{n,t-y}[\Omega^z]$ (cf. paragraphe 3.3). Ainsi, en combinant $\mathcal{T}$ et l'algorithme décrit dans la Figure 3.5, nous obtenons un algorithme qui résout le (n, z)-accord dans le modèle $\mathcal{MP}_{n,t-y}[\diamond\mathcal{S}_x]$. Par conséquent, d'après le théorème 3.2, les paramètres t, y, x et z vérifient la relation $t - y < z + x - 1$, i.e., $t + 1 < x + y + z$, ce qui contredit l'hypothèse initiale. $\square_{\textit{Théorème 3.3}}$

Le corollaire suivant est une conséquence immédiate de la démonstration du théorème 3.3.

Corollaire 3.4 *Dans les modèles $\mathcal{MP}_{n,t}[\diamond\mathcal{S}_x, \psi^y]$, $\mathcal{MP}_{n,t}[\diamond\mathcal{S}_x, \diamond\psi^y]$ et $\mathcal{MP}_{n,t}[\mathcal{S}_x, \diamond\psi^y]$, il existe un algorithme qui construit un détecteur de défaillances de la classe Ω^z si et seulement si $(x + y + z) > t + 1$.*

En observant qu'un détecteur $\diamond\mathcal{S}_1$ ne fournit aucune information sur les défaillances, nous déduisons de l'algorithme des « deux roues » et du théorème 3.3 le corollaire suivant :

Corollaire 3.5 *Il existe un algorithme qui implémente un détecteur de défaillances de la classe Ω^z dans les modèles $\mathcal{MP}_{n,t}[\psi^y]$ ou $\mathcal{MP}_{n,t}[\diamond\psi^y]$ si et seulement si $y + z > t$.*

De même, puisqu'un détecteur $\diamond\psi^0$ ne fournit aucune information sur les défaillances, nous avons :

Corollaire 3.6 *Il existe un algorithme qui implémente un détecteur de défaillances de la classe Ω^z dans les modèles $\mathcal{MP}_{n,t}[\mathcal{S}_x]$ ou $\mathcal{MP}_{n,t}[\diamond\mathcal{S}_x]$ si et seulement si $x+z > t+1$.*

Relations entre les classes $\mathcal{S}_x/\diamond\mathcal{S}_x$ et $\psi^y/\diamond\psi^y$ Pour compléter la grille, nous étudions dans cette partie les relations entre les classes $\mathcal{S}_x/\diamond\mathcal{S}_x$ et $\psi^y/\diamond\psi^y$. Nous démontrons qu'en général ces classes sont incomparables, c'est à dire que dans le modèle $\mathcal{MP}_{n,t}[\mathcal{S}_x/\diamond\mathcal{S}_x]$ (respectivement $\mathcal{MP}_{n,t}[\psi^y/\diamond\psi^y]$), il n'existe pas d'algorithme qui implémente un détecteur de défaillances $\psi^y/\diamond\psi^y$ (respectivement $\mathcal{S}_x/\diamond\mathcal{S}_x$).

Théorème 3.4 *Soient x, y tels que $1 \leq x \leq t+1$ et $1 \leq y \leq t$. Il est impossible de construire un détecteur $\diamond\psi^y$ (ou ψ^y) dans les modèles $\mathcal{MP}_{n,t}[\diamond\mathcal{S}_x]$ ou $\mathcal{MP}_{n,t}[\mathcal{S}_x]$.*

Démonstration Nous considérons le modèle « le plus fort » ($\mathcal{MP}_{n,t}[\mathcal{S}_x]$) et nous démontrons qu'il n'existe pas d'algorithme qui implémente le détecteur le « plus faible » $\diamond\psi^y$. La démonstration reste valide si l'on considère le modèle $\mathcal{MP}_{n,t}[\diamond\mathcal{S}_x]$ car $\mathcal{S}_x \subseteq \diamond\mathcal{S}_x$. De même, nous obtenons immédiatement l'impossibilité de construire ψ^y car $\psi^y \subseteq \diamond\psi^y$.

Pour simplifier la preuve, nous utilisons la classe $\diamond\phi^y$. La démonstration procède par contradiction. Supposons qu'il existe un algorithme $\mathcal{A}$ qui construise un détecteur $\diamond\phi^y$ dans le modèle $\mathcal{MP}_{n,t}[\diamond\mathcal{S}_x]$. Nous exhibons une exécution e dans laquelle la propriété de sûreté n'est pas satisfaite

Soit $E \subseteq \Pi$, $|E| = t - y + 1$ et $E \cap Correct \neq \emptyset$. Soit p_c un processus correct qui n'appartient pas à l'ensemble E. De plus, on suppose que le détecteur $\mathcal{S}_x$ ne soupçonne jamais p_c dans l'exécution e (c'est à dire $\forall\tau \geq 0, \forall i : c \in \text{TRUSTED}_i^\tau$). Un tel processus existe d'après la propriété de précision perpétuelle de la classe $\mathcal{S}_x$. Soit τ_0 un instant à partir duquel toutes les invocations QUERY(E) retournent la valeur *false*. Un tel instant existe puisque l'algorithme $\mathcal{A}$ implémente correctement un détecteur $\diamond\psi^y$, et donc en particulier sa propriété de sûreté inéluctable. Nous considérons deux exécutions $e1$ et $e1'$ définies comme suit :

- Les exécutions $e1$ et e sont indistinguables par les tous les processus jusqu'à l'instant τ_0. A l'instant $\tau_0 + 1$, tous les processus qui sont dans l'ensemble E défaillent. Soit $\tau_1 > \tau_0$ un instant auquel un processus $p_i, i \in \Pi - E$ invoque QUERY(E) et obtient la valeur *true*. L'existence d'un tel instant est garantie par la propriété de vivacité de la classe $\diamond\phi^y$.
- Les exécutions $e1'$ et e sont indistinguables par tous les processus jusqu'à l'instant τ_0. Dans l'exécution $e1'$, tous les processus dans l'ensemble E sont corrects, mais tous leurs messages envoyés entre les instants $\tau_0 + 1$ et τ_1 sont retardés jusqu'à l'instant $\tau_1 + 1$. Ceci signifie qu'un message m, envoyé par un processus $p_j, j \in E$ entre les instants $\tau_0 + 1$ et τ_1 est reçu par son destinataire au plus tôt à l'instant $\tau_1 + 1$. L'asynchronie du système autorise une durée arbitraire de transmission des messages.

De plus, dans les deux exécutins $e1$ et $e1'$, pour chaque processus, les sorties du détecteur $\mathcal{S}_x$ sont les exactement les mêmes entre les instants 0 et τ_1. Remarquons que, quelque soient les sorties du détecteur de la classe $\mathcal{S}_x$ dans l'exécution $e1$, les sorties dans l'exécution $e1'$ peuvent être exactement les mêmes sans violer les propriétés de la classe $\mathcal{S}_x$. La précision limitée perpétuelle est satisfaite car le processus p_c est correct et n'est jamais soupçonné dans $e1$ et $e1'$. Puisque la complétude forte est une propriété « inéluctable » (un certain prédicat doit être verifié de façon permanenente à partir d'un certain instant arbitraire), elle est toujours satisfaite dans n'importe quel préfixe fini de n'importe quelle exécution.

Il est clair que, jusqu'à l'instant τ_1, les processus qui appartiennent à l'ensemble $\Pi - E$ sont dans l'incapacité de distinguer l'exécution $e1$ et de l'exécution $e1'$. On en déduit que dans $e1'$, une invocation de QUERY(E) à l'instant τ_1 par le processus p_i retourne la valeur *true*. Cependant, dans l'exécution $e1'$, à partir de l'instant $\tau_1 > \tau_0$,

la propriété de sûreté est satisfaite : l'invocation devrait retourner la valeur *false* car E contient un processus correct : une contradiction. $\square_{Théorème\ 3.4}$

Théorème 3.5 *Soient $0 \leq y < t$ et $1 < x \leq t+1$. Dans les modèles $\mathcal{MP}_{n,t}[\diamond\psi^y]$ et $\mathcal{MP}_{n,t}[\psi^y]$, construire un détecteur $\diamond\mathcal{S}_x$ ou $\mathcal{S}_x$ est impossible.*

Démonstration Pour établir le résultat, nous considérons la plus faible classe non triviale ($\diamond\mathcal{S}_2$) et nous démontrons qu'il n'existe pas d'algorithme qui construit un détecteur de cette classe dans le modèle le plus fort ($\mathcal{MP}_{n,t}[\psi^{t-1}\ ]$).

Supposons qu'il existe un algorithme $\mathcal{A}$ qui simule un détecteur $\diamond\mathcal{S}_2$ dans le modèle $\mathcal{MP}_{n,t}[\psi^{t-1}]$. Dans les exécutions dans lesquelles le nombre de défaillances est 0 ou 1, la sortie du détecteur ψ^{t-1} est toujours 1. À partir de cette observation, nous construisons une exécution infinie E de l'algorithme $\mathcal{A}$ dans laquelle la propriété de *2-représentant commun inéductable* n'est jamais satisfaite (cf. définition 3.1) . Intuitivement, le détecteur ψ^{t-1} n'est pas assez fin pour discerner une exécution où tous les processus sont corrects d'une exécution dans laquelle il existe un processus défaillant. Nous exploitons cette observation pour construire une exécution dans laquelle il existe un processus dont le représentant change infiniment souvent.

E est construite en répétant procédé ci-dessous. Chaque itération i construit un préfixe E_i de plus en plus grand de E qui étend le préfixe précédent E_{i-1}. Par ailleurs, dans l'extension générée à l'étape $i > 0$, il existe deux instant τ_i et $\tau_{i'}$ et un processus $p_{(i)}$ tel que $\text{REPR}_{(i)}^{\tau_i} \neq \text{REPR}_{(i)}^{\tau'_i}$.

Pour débuter, on choisit une exécution infinie arbitraire dans laquelle tous les processus sont corrects. E_0 est un préfixe arbitraire de cette exécution.

Supposons construit un préfixe E_{i-1} de E. On construit une extension E_i de la façon suivante.

- R_1 est une exécution infinie dans laquelle tous les processus sont corrects et telle que E_i est un préfixe de R_1. On note $\tau_{f(i-1)}$ le dernier instant de E_{i-1}. Par hypothèse, l'algorithme $\mathcal{A}$ simule correctement un détecteur $\diamond\mathcal{S}_2$. Dans R_1 il existe donc un instant $\tau_s > \tau_{f(i-1)}$, un ensemble X_i d'au moins deux processus et une identité ℓ_i tel que, à l'instant τ_s, la propriété de 2-représentant commun est satisfaite. Plus précisément, on a : $|X_i| \geq 2$; $\forall\alpha \notin X_i$: $\text{REPR}_\alpha^{\tau_s} = \alpha$ et $\forall\alpha \in X_i : \text{REPR}_\alpha^{\tau_s} = \ell_i$.
- Soit R_2 une exécution identique à R_1 jusqu'à l'instant τ_s. A l'instant $\tau_s + 1$, p_{ℓ_i} tombe en panne ; les autres processus sont corrects. Le comportement des processus et les instants de réception des messages sont arbitraires après τ_s.
  L'algorithme $\mathcal{A}$ est correct : dans R_2, la propriété de représentant 2-commun est donc satisfaite à partir d'un certain instant $\tau'_s > \tau_s + 1$. Nous notons X'_i et ℓ'_i un couple (ensemble, identité) qui satisfont cette propriété à l'instant τ'_s. Il est clair que $\ell_i \neq \ell'_i$.
- Le préfixe E_i est défini sur l'intervalle $[0, \tau'_s]$. Pour tous les processus à l'exception de p_{ℓ_i}, E_i est identique à l'exécution R_2 sur l'intervalle $[0, \tau'_s]$. C'est à dire que, durant l'intervalle $[0, \tau'_s]$, ces processus effectuent les mêmes étapes de calcul aux

mêmes instants dans E_i et R_2. De même, les instants de réception des messages sont identiques.
Pour le processus p_{ℓ_i}, les exécutions E_i et R_2 sont identiques jusqu'à l'instant τ_s. Dans l'intervalle $[\tau_s, \tau'_s]$, le processus p_{ℓ_i} n'effectue aucune action. Ce comportement est légal car le système est asynchrone : nous pouvons « geler » pour une durée finie arbitraire l'activité d'un processus.

- Pour chaque processus, la sortie du détecteur de la classe ψ^{t-1} est identique dans R_2 et E_i (elle est toujours égale à 1). Par conséquent, jusqu'à τ'_s, un processus $p_\alpha \neq p_{\ell_i}$ ne peut distinguer l'exécution E de E'. A l'instant τ'_s, la sortie de l'algorithme $\mathcal{A}$ est donc la même à chaque processus $p_j \neq p_{\ell_i}$ dans R_2 et E_i.
- Finalement, soit p_α un processus tel que $\alpha \in X_i \wedge \alpha \neq \ell_i$. Dans R_2, $\text{REPR}_\alpha^{\tau'_s} \neq \ell_i$ car p_{ℓ_i} est fautif et par choix de τ'_s, la propriété de représentant 2-commun est satisfaite à partir cet instant. Donc, $\text{REPR}_\alpha^{\tau'_s} \neq \ell_i$ dans E_i. Cependant dans E_i, à l'instant τ_s, $\text{REPR}_\alpha^{\tau} = \ell_i$. De plus, l'intervalle $[\tau_s, \tau'_s]$ est inclus E_i et n'est pas inclus dans E_{i-1}.

Dans E_i, il existe bien deux instants $\tau_i (= \tau_s)$, $\tau'_i (= \tau'_s)$ et un processus $p_{(i)} (= p_\alpha)$ tels que $\text{REPR}_{(i)}^{\tau_i} \neq \text{REPR}_{(i)}^{\tau'_i}$. De plus l'intervalle $[\tau_i, \tau'_i]$ est inclus dans $E_i - E_{i-1}$.

L'itération du procédé décrit ci-dessus construit une exécution infinie du modèle $\mathcal{MP}_{n,t}[\psi^{t-1}\,]$ dans laquelle tous le processus sont corrects. Cette exécution possède la propriété suivante : $\forall \tau : \exists \tau_1, \tau_2, p_i$ tels que $(\tau \leq \tau_1 < \tau_2) \wedge (\text{REPR}_i^{\tau_1} \neq \text{REPR}_i^{\tau_2})$. Comme le nombre de processus est fini, il existe donc un processus p_i tel que la valeur de REPR_i change infiniment souvent. L'algorithme $\mathcal{A}$ n'implémente donc pas un détecteur de la classe $\diamond\mathcal{S}_2$. □*Théorème* 3.5

De Ω^z vers $\mathcal{S}_x/\diamond\mathcal{S}_x$ ou $\psi^y/\diamond\psi^y$ Nous avons montré (Corollaires 3.5 et 3.6) qu'il est possible de construire un détecteur Ω^z dans le modèle $\mathcal{MP}_{n,t}[\mathcal{S}_x/\diamond\mathcal{S}_x]$ (respectivement $\mathcal{MP}_{n,t}[\psi^y/\diamond\psi^y]$) si et seulement si $x + z > t + 1$ (respectivement, $y + z > t$). Dans ce paragraphe, nous établissons l'impossibilité des transformations inverses, c'est à dire qu'il est impossible de construire un détecteur $\mathcal{S}_x/\diamond\mathcal{S}_x$ (respectivement, $\psi^y/\diamond\psi^y$) dans le modèle $\mathcal{MP}_{n,t}[\Omega^z]$. Ces résultats se déduisent des théorèmes 3.4 et 3.5.

Corollaire 3.7 *Soient $1 \leq y \leq t$ et $1 \leq z \leq t + 1$. Il n'existe pas d'algorithme qui implémente un détecteur $\psi^y/\diamond\psi^y$ dans le modèle $\mathcal{MP}_{n,t}[\Omega^z]$.*

Démonstration La démonstration procède par contradiction. Supposons qu'il existe un algorithme $\mathcal{A}$ qui construise un détecteur $\diamond\psi^y$ dans le modèle $\mathcal{MP}_{n,t}[\Omega^z]$. On sait qu'il existe un algorithme $\mathcal{B}$ qui construit un détecteur Ω^z dans le modèle $\mathcal{MP}_{n,t}[\diamond\mathcal{S}_x]$ lorsque $x + z > t + 1$ (corollaire 3.6). En combinant $\mathcal{A}$ et $\mathcal{B}$, on obtient un algorithme qui implémente un détecteur $\diamond\psi^y$, $1 \leq y \leq t$ dans le modèle $\mathcal{MP}_{n,t}[\diamond\mathcal{S}_x]$. L'existence d'un tel algorithme contredit le théorème 3.4. □*Corollaire* 3.7

Corollaire 3.8 *Soit $1 < x, z \leq t$. Il n'existe pas d'algorithme qui implémente un détecteur $\mathcal{S}_x/\diamond\mathcal{S}_x$ dans le modèle $\mathcal{MP}_{n,t}[\Omega^z]$.*

Démonstration La démonstration est similaire à la démonstration précédente. Elle est laissée en exercice. $\square_{Corollaire\ 3.8}$

Résumé

Pour un type d'objet O, soit $scn(O)$ le plus entier k tel qu'il existe un algorithme qui résout le (n,k)-accord dans le modèle $\mathcal{MP}_{n,t}[O]$, c'est à dire dans le modèle à passage de messages muni d'objets de type O. $scn(O)$ mesure la capacité du type O à résoudre le (n,k)-accord. Étant donnés deux types d'objets $O1$ et $O2$ tels que $scn(O1) = k1$ et $scn(O2) = k2$, est-il possible de les combiner pour résoudre le (n,k)-accord avec $k < \min(k1, k2)$? En d'autres termes, la hiérarchie induite par la fonction scn est-elle robuste ?

Dans ce chapitre, nous avons étudié de cette question lorsque les objets sont des détecteurs de défaillances $\mathcal{C}$ appartenant aux familles $(\mathcal{S}_x/\diamond\mathcal{S}_x)_{1\leq x\leq n}$, $(\psi^y/\diamond\psi^y)_{1\leq y\leq n}$ ou $(\Omega^z)_{1\leq z\leq n}$. La valeur de $scn(\mathcal{C})$ pour $\mathcal{C} \in (\mathcal{S}_x/\diamond\mathcal{S}_x)_{1\leq y\leq n}$ a été établie dans [71]. Les détecteurs des familles $(\psi^y/\diamond\psi^y)_{1\leq y\leq n}$ introduites dans ce chapitre fournissent une estimation plus ou moins précise du nombre de défaillances. Ce type d'information peut être vue comme une généralisation de l'hypothèse classique d'une borne t sur le nombre de défaillances. Nous montrons l'équivalence entre les classes $\psi^y/\diamond\psi^y$ et les classes $\phi^y/\diamond\phi^y$ introduites dans [90].

Nous établissons ensuite la valeur de $scn(\mathcal{C})$ pour $\mathcal{C}$ appartenant à $(\psi^y/\diamond\psi^y)_{1\leq y\leq n}$ et $(\Omega^z)_{1\leq z\leq n}$: $scn(\psi^y/\diamond\psi^y) = t-k+1$ et $scn(\Omega^z) = z$.

Le résultat principal de ce chapitre est le théorème « $\diamond\mathcal{S}_x + \diamond\psi^y \rightsquigarrow \Omega^z$ si et seulement si $x+y+z > t+1$ ». Ainsi, dans un environnement asynchrone où le nombre de défaillances est borné par t, il est possible de combiner des détecteurs $\diamond\mathcal{S}_x$ ($scn(\diamond\mathcal{S}_x) = t+2-x$) et $\diamond\psi^y$ ($scn(\diamond\psi^y) = t+1-y$) pour construire un détecteur de la classe Ω^z tel que $z = (t+2-x)-y = (t+1-y)-(x-1)$. La hiérarchie induite par la « capacité à résoudre le (n,k)-accord » n'est donc pas robuste. L'algorithme qui réalise l'addition met en œuvre un principe extrêmement simple :« tester toutes les configurations possibles jusqu'à identification d'une configuration correcte ». La difficulté provient du fait que la correction d'une configuration dépend de deux propriétés sous-jacentes inéluctables, c'est à dire vérifiées à partir d'un certain temps initialement non connu.

Enfin, nous avons brossé un tableau complet des relations liant les détecteurs des familles $(\mathcal{S}_x/\diamond\mathcal{S}_x)_{1\leq x\leq n}$, $(\psi^y/\diamond\psi^y)_{1\leq y\leq n}$ et $(\Omega^z)_{1\leq z\leq n}$. Pour chaque couple de détecteurs $(\mathcal{C}1, \mathcal{C}2)$ appartenant à ces familles, nous avons donné un algorithme fondé sur $\mathcal{C}1$ qui implémente $\mathcal{C}2$ ou montré l'impossibilité d'une telle transformation.

Autres détecteurs orientés vers le (n,k)-accord Dans des travaux postérieurs à la publication de [92], plusieurs équipes de recherche ont introduit de nouvelles classes de détecteurs dans l'optique d'identifier le détecteur le plus faible qui permet de résoudre le (n,k)-accord. Ces classes sont strictement plus faibles que la classe Ω^k (la plus faible classe qui permet de résoudre ce problème parmi les classes étudiées dans ce chapitre).

Le détecteur Υ défini par Herlihy et coauteurs [61] fournit à chaque processus un ensemble d'identités qui se stabilise inéluctablement sur un certain ensemble T. L'information sur les défaillances est très faible puisque la seule contrainte est que $T \neq \mathit{Correct}$. Ce détecteur est affaibli dans les travaux de Chen et coauteurs dans [34, 35, 61]. L'approche développée dans ces travaux consiste à découper le systèmes en sous-ensembles $S_1, \ldots, S_\alpha$ deux à deux disjoints. Chaque sous-système est équipé d'un détecteur $\mathcal{C}$, cependant seul un ensemble S_i bénéficie des pleines capacités du détecteur $\mathcal{C}$. Dans les autres sous-systèmes, seule une partie des propriétés de $\mathcal{C}$ sont satisfaites. Cette ligne de recherche peut être vue comme le dual de la question étudiée dans ce chapitre. En effet, si nous cherchons à combiner des détecteurs existants pour obtenir un détecteur plus puissant, [35] explore la voix opposée : à partir de détecteurs existants, comment dériver des familles de détecteur strictement plus faibles ? Enfin, Zieliński [124, 123, 125] définit la classe anti-Ω et montre que cette classes est nécessaire et suffisante pour résoudre le $(n, n-1)$-accord.

Mise en œuvre des détecteurs de défaillances Dans le contexte des détecteurs de défaillances, un aspect important est la question de leur mise en œuvre. En effet, un oracle détecteur de défaillances abstrait des hypothèses de bas niveau sur le système (par exemple : des garanties de temps livraison de messages sur certains canaux de communication ou des temps de réponses de type « *ping* » plus rapides de certains sites). Cependant, la définition d'une classe de détecteurs ne donne aucune indication sur les propriétés concrètes du système. La recherche de la mise en œuvre des détecteur vise donc à formuler des hypothèses concrètes, notamment de synchronie, sur le comportement du systèmes et à concevoir un algorithme $\mathcal{A}$ qui, lorsque ces hypothèses sont satisfaites, implémente un détecteur donné. Une suite de papiers présente des hypothèses de plus en plus faibles qui permettent d'implémenter le détecteur Ω ou $\diamond\mathcal{S}$ [30, 13, 87, 8, 9, 97, 99, 83, 46, 112].

Le théorème d'addition montre que du point de vue « détecteurs de défaillances », les informations fournies par les détecteurs $\diamond\psi^y$ et $\diamond\mathcal{S}_x$ ne sont pas de même nature. Cette remarque laisse supposer l'existence d'une famille d'hypothèses, orthogonale à celles introduites pour mettre en œuvre $\diamond\mathcal{S}$ ou Ω, qui autorise la construction de $\diamond\psi^y$. L'existence d'une telle famille ouvrirai la voie vers la conception d'algorithmes implémentant Ω qui bénéficieraient d'une grande couverture d'hypothèses [108]. Plus précisément, soient $(P(x))$ et $(Q(y))$ deux familles d'hypothèses orthogonales qui permettent respectivement d'implémenter $\diamond\mathcal{S}_x$ et $\diamond\psi^y$. L'orthogonalité s'entend ici dans le sens suivant : si l'on suppose que la propriété $P(x)$ est satisfaite, cette hypothèse n'implique pas que la propriété $Q(y)$ est vérifiée et réciproquement. L'objectif serait de concevoir un algorithme qui « fait de son mieux » (*best effort*) pour implémenter Ω dans le modèle asynchrone. De plus, la sortie de cet algorithme doit satisfaire la propriété de la classe Ω dès que le système satisfait les hypothèses $P(x)$ et $Q(y)$ avec $x + y + 1 > t + 1$. [103] constitue un premier pas dans la recherche d'hypothèses $P(y)$.

Chapitre 4

Vers une caractérisation algorithmique des détecteurs de défaillances

Les détecteurs de défaillances restreignent l'asynchronie du modèle. Ainsi, l'addition d'un détecteur Ω^z, $\diamond\mathcal{S}_x$ ou $\diamond\psi^y$ permet de résoudre le k-accord, comme nous l'avons vu dans la chapitre précédent. Dans le cas du détecteur Ω^z, en faisant varier z de n à 1, nous obtenons des modèles de plus en plus synchrones dans lesquels il est possible de résoudre des problèmes de plus en plus contraints (pour chaque z, le modèle augmenté permet de résoudre le z-accord). Bien que ces détecteurs possèdent la même puissance vis à vis du k-accord, ils ne sont pas équivalents (cf. paragraphe 3.5). Les résultats de réductions/impossibilité amènent à se demander comment caractériser la « quantité » et le « type » de synchronie introduite par l'addition d'un détecteur de défaillances dans le modèle asynchrone. C'est principalement cette question que nous étudions dans ce chapitre, dans le contexte des familles $(\Omega^z)_{1\leq z\leq n}$, $(\diamond\mathcal{S}_x)_{1\leq x\leq n}$ et $(\diamond\psi^y)_{0\leq y\leq n-1}$.

De nombreux travaux ont investigué cette question dans le contexte de modèles asynchrones enrichis avec des détecteurs qui offrent une puissance suffisante pour résoudre le consensus. L'approche fréquemment employée consiste à enrichir le modèle avec des propriétés synchrones (sur les délais de transfert des messages, les vitesses relatives des processus, etc.) pour ensuite concevoir des algorithmes qui mettent en œuvre ces détecteurs (voir par exemple [9, 83, 46, 99]). D'autres articles [50, 78] définissent des modèles dans lesquels le calcul procède par rondes asynchrones qui englobent le détecteur sous-jacent. En particulier, le travail présenté dans [50] vise à définir un modèle général qui permet d'exprimer de façon unifiée certaines abstraction qui restreignent l'asynchronie (détecteurs de défaillances ou hypothèses de synchronie partielle). La puissance du modèle augmenté par l'une de ces abstractions est caractérisée par un prédicat sur l'ensemble des messages reçus qui indique à chaque processus quand il peut passer à la ronde suivante. Autrement dit, ce prédicat détermine la « quantité d'information » maximale qu'un processus peut accumuler en fonction des propriétés de l'abstraction sous-jacente. Cette approche formalise l'intuition que plus un modèle est puissant, moins l'incertitude

locale sur l'état global est grande.

Dans la lignée de ces travaux, nous choisissons d'étudier la puissance apportée par les détecteurs de défaillances par l'intermédiaire du modèle *snasphot immédiat itéré* (noté *IIS* et brièvement introduit dans le chapitre 1) dans lequel les processus accèdent à chaque ronde à un nouvel objet snapshot immédiat. Les exécutions dans ce modèle ont l'avantage par rapport aux autres modèles fondés sur le calcul par rondes d'être hautement structurées, simplifiant leur analyse. De plus, dans le cas de modèles purement asynchrones, le modèle *IIS* et le modèle traditionnel à registres sont équivalents du point de vue de la calculabilité. La démonstration élégante du théorème de calculabilité asynchrone de Borowsky et Gafni [24] repose sur ce modèle.

Contributions Nous montrons comment caractériser la synchronie apportée par les détecteurs introduits dans le chapitre précédent au travers d'une extension du modèle *IIS*. Pour capturer la synchronie du modèle à registres équipé de détecteurs, nous considérons des restrictions de ce modèle.

Plus précisément, nous présentons les résultats suivants. $\mathcal{C}$ désigne un détecteur appartenant à l'une des familles $(\Omega^z)_{1\leq z\leq n}$, $(\diamond\mathcal{S}_x)_{1\leq x\leq n}$ ou $(\diamond\psi^y)_{0\leq y\leq n}$.

1. Étant donné un détecteur $\mathcal{C}$, nous définissons un modèle *IIS* restreint qui lui correspond. Ce modèle sera noté $IRIS(PR_{\mathcal{C}})$. Chaque exécution dans le modèle $IRIS(PR_{\mathcal{C}})$ est une exécution valide dans le modèle *IIS*. Par contre, l'inverse n'est pas nécessairement vrai. Pour cela, nous contraignons la spécification du snapshot immédiat. Le procédé est similaire à celui employé dans le paragraphe 2.3.2 du chapitre 2, où nous avons vu comment « embarquer » les propriétés du k-test&set dans un objet snapshot immédiat plus contraint. La puissance offerte par des objets k-test&set se traduit par l'ajout de la propriété de simultanéité bornée dans la spécification du snapshot immédiat. Ici, la puissance du détecteur $\mathcal{C}$ est capturée par une propriété $PR_{\mathcal{C}}$ appropriée que l'on additionne à la spécification des objets snapshot immédiats.
2. Nous montrons ensuite que la restriction $IRIS(PR_{\mathcal{C}})$ caractérise la synchronie du modèle à registres équipé d'un détecteur $\mathcal{C}$. Pour obtenir ce résultat, nous présentons dans un premier temps une série d'algorithmes qui simulent le modèle $IRIS(PR_{\mathcal{C}})$ dans le modèle à registre muni d'un détecteur $\mathcal{C}$. Réciproquement, nous montrons comment extraire un détecteur $\mathcal{C}$ dans le modèle $IRIS(PR_{\mathcal{C}})$. Un corollaire remarquable de ces transformations est l'équivalence entre le modèle à registre équipé de $\mathcal{C}$ et le modèle itéré $IRIS(PR_{\mathcal{C}})$ associé du point de vue de la calculabilité d'un sous ensemble des problèmes de décision. Plus précisément, nous établissons que pour tout problème *d'accord*[1], il existe une solution $(n-1)$-résiliente dans le modèle $\mathcal{SM}_{n,n-1}[\mathcal{C}]$ ([2]) si et seulement si il existe une

[1] Un problème d'accord est un problème de décision dont la spécification implique la propriété suivante : tout processus qui connaît une décision légale d'un autre processus peut localement calculer sa propre décision.

[2] Rappelons que la notation $\mathcal{SM}_{n,n-1}[\mathcal{C}]$ désigne le modèle asynchrone à registres composé de n processus. Le nombre de défaillance est bornée par $n-1$ dans toute exécution ($t=n-1$). Les processus ont accès à un détecteur de défaillances $\mathcal{C}$.

solution dans le modèle associé $IRIS(PR_{\mathcal{C}})$. La démonstration repose sur un algorithme qui simule tout algorithme écrit pour le modèle $\mathcal{SM}_{n,n-1}[\mathcal{C}]$ dans le modèle $IRIS(PR_{\mathcal{C}})$. La technique algorithmique étend la simulation développée dans [24].

3. Enfin, en application les résultats et les outils développés précédemment, nous présentons de nouvelles preuves simples de l'impossibilité de résoudre le k-accord à l'aide de ces détecteurs.

Organisation du chapitre Le chapitre suit le plan suivant :

- Le paragraphe 4.1 rappelle la définition du modèle *IIS* et ses propriétés.
- Les modèles restreints $IRIS(PR_{\Omega^z})$, $IRIS(PR_{\diamond\mathcal{S}_x})$ et $IRIS(PR_{\diamond\psi^y})$ sont définis dans le paragraphe 4.2. Une série de transformations qui montrent pour chaque détecteur $\mathcal{C} \in \{\Omega^z, \diamond\mathcal{S}_x, \diamond\psi^y\}$ comment simuler le modèle correspondant $IRIS(PR_{\mathcal{C}})$ est ensuite présentés.
- Réciproquement, le paragraphe 4.3 couvre les transformations inverses : il montre comment extraire un détecteur $\mathcal{C}$ dans le modèle $IRIS(PR_{\mathcal{C}})$.
- Le paragraphe 4.4 étend la simulation développée dans [24, 23] pour établir l'équivalence pour les problèmes d'accord entre les modèles $IRIS(PR_{\mathcal{C}})$ et $\mathcal{SM}_{n,n-1}[\mathcal{C}]$.
- Enfin, dans le paragraphe 4.5 nous appliquons les résultats précédents à l'étude d'une famille de détecteurs faibles qui combinent les caractéristiques des détecteurs $\diamond\mathcal{S}_x$ et Ω^z. Pour chaque détecteur de cette famille, nous déterminons le plus petit k pour lequel il existe une solution fondée sur ce détecteur au problème du (n,k)-accord. La démonstration est relativement simple et repose uniquement sur des réductions algorithmiques. Ceci contraste avec la seule démonstration connue d'un résultat similaire établi pour la famille $(\diamond\mathcal{S}_x)_{1\leq x\leq n}$ par Herlihy et Penso [71] qui repose sur des techniques empruntées à la topologie algébrique.

4.1 Le modèle *IIS*

Dans ce paragraphe, nous décrivons le modèle *IIS* ainsi que certaines de ses propriétés. Nous commençons par rappeler la spécification des objets de base du modèle, à savoir les objets *snapshot immédiat*.

Snapshot immédiat à usage unique Comme nous l'avons vu dans le paragraphe 1.3, un objets *snapshot immédiat* (notés *IS*) est une abstraction de haut niveau qui facilite la conception d'algorithmes. Un tel objet exporte une unique primitive write_snap() qui permet en une seule opération de modifier et de lire le contenu de l'objet. Plus précisément, lorsque p_i invoque write_snap() sur un objet *IS*, il obtient en retour un *snapshot* ou *vue* noté sm_i qui est l'ensemble des valeurs écrites précédemment ou concurremment par les autres processus. Si l'on suppose que chaque processus p_i invoque IS.write_snap(i), alors les snapshots sm_i retournés satisfont collectivement les propriétés suivantes (cf. définition 1.4)

- Auto inclusion : $\forall i : i \in sm_i$;
- Comparaison : $\forall i, j : (sm_i \subseteq sm_j) \vee (sm_j \subseteq sm_i)$;

– Immédiateté : $\forall i, j : i \in sm_j \Rightarrow sm_i \subseteq sm_j$.

Dans le modèle *IIS*, les objets snapshot immédiat sont à usage unique. Ceci signifie que chaque processus ne peut qu'exécuter au plus une opération write_snap() sur chacun de ces objets.

Le modèle snapshot immédiat itéré Dans le modèle Snapshot Immédiat Itéré (*IIS*), les processus communiquent exclusivement via des objets snapshot immédiat à usage unique (*IS*). Le médium de communication est composé d'une infinité d'objets snapshot immédiats $IS[1], IS[2], \ldots$. Une exécution dans ce modèle se déroule par rondes asynchrones consécutives. A chaque ronde, les processus accèdent à un nouvel objet snapshot immédiat, en suivant le motif suivant :

$r_i \leftarrow 0$;
loop forever $r_i \leftarrow r_i + 1$;
local computations ; compute v_i ;
$sm_i \leftarrow IS[r_i]$.write_snap$(< i, v_i >)$;
local computations
end loop.

Par rapport au modèle classique dans lequel les processus accèdent de façon répétée à un même objet *IS*, les vues successives obtenues par les processus ont dans le modèle une structure récursive régulière. Cette régularité est la base de l'élégante caractérisation des problèmes de décision pour lesquels il existe une solution sans attente dans le modèle à registre présentée dans [24]. Par exemple, les Figures 4.1 et 4.2 décrivent toutes les vues possibles après respectivement une ronde et deux rondes.

Il est parfois plus facile de se représenter les vues obtenues au cours d'une ronde comme un certaine répartition des processus sur les marches de l'échelle de Borowsky Gafni (voir paragraphes 1.3.3 et 2.3.2). La figure décrit la répartition des processus dans les deux rondes qui correspondent à l'exécution représentée par le triangle grisé dans la Figure 4.2.

Enfin, la Figure 4.4 présente un autre point vue sur cette exécution. Pour un observateur extérieur, les opérations write_snap() semblent se dérouler instantanément. Il est possible de positionner ces instants sur un axe de temps commun (plusieurs opérations sont placées au même point – un objet snapshot immédiat est set-linéarisable [105]). Ainsi, lors de la deuxième ronde, p_1 exécute son opération tout seul et avant les deux autres processus. Il est *ordonnancé* avant p_2 et p_3. Par contre, les opérations de p_2 et p_3 sont concurrentes et retournent le même snapshot. Ceci est matérialisé dans la Figure 4.4 par un seul point et par le placement de ces deux processus sur la même marche dans la Figure 4.3.

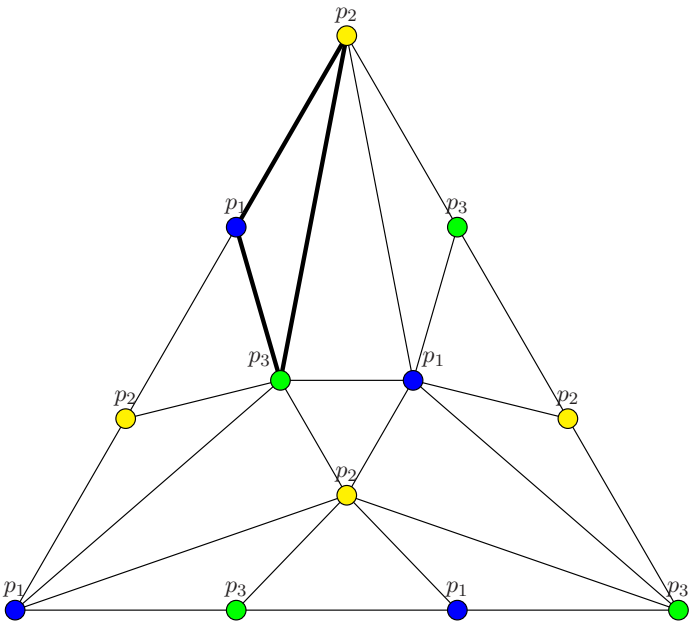

FIG. 4.1 – 3 processus, 1 ronde : toutes les exécutions

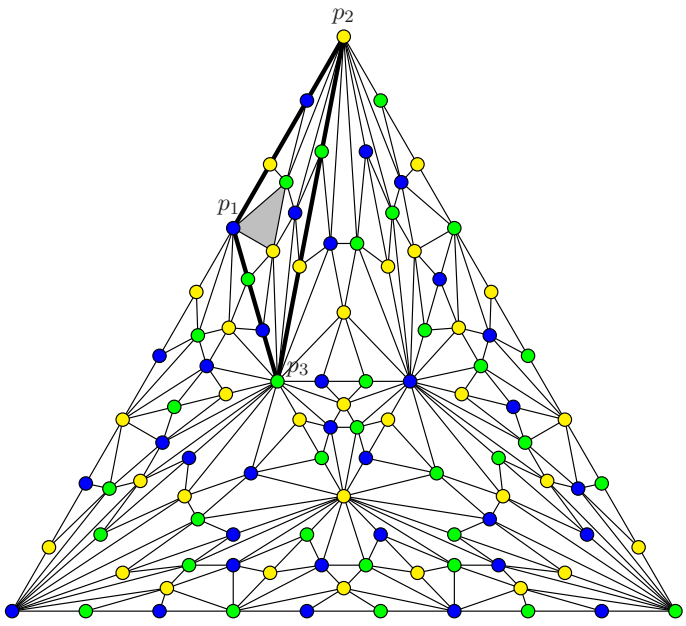

FIG. 4.2 – 3 processus, 2 rondes : toutes les exécutions

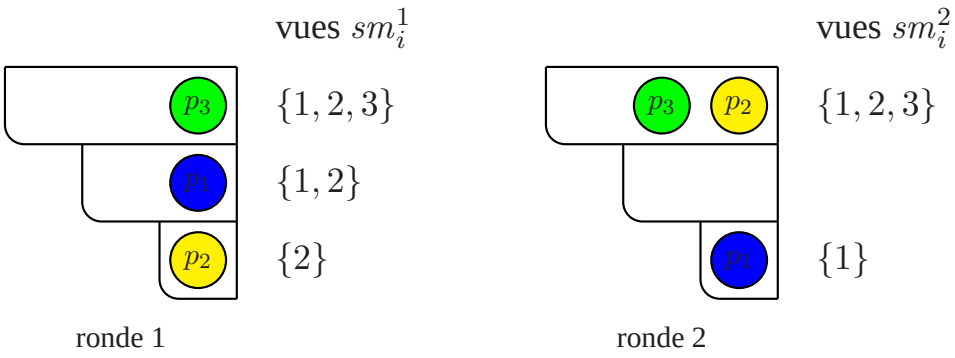

FIG. 4.3 – Ordonnancement des processus sur les marches

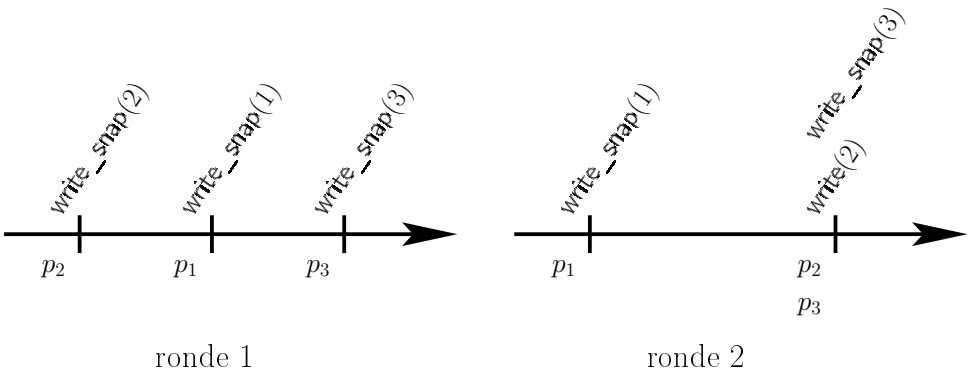

FIG. 4.4 – Set linéarisation

4.1.1 Notion de processus corrects dans le modèles *IIS*

Dans le modèle à registres ou muni d'objets snapshot accessibles de façon répétée, la communication entre deux processus corrects est fiable. Si le processus correct p_i écrit v en mémoire partagée et n'écrase pas ensuite cette valeur, alors tout processus correct p_j qui lit régulièrement la mémoire observera inéluctablement v.

Différemment, dans le modèle *IIS*, il est possible que p_j n'observe jamais v, même si p_i effectue une infinité d'opérations write_snap(v). L'existence de ce type d'exécutions découle de la contrainte imposée par le modèle sur l'usage des objets *IS*. Cette situation se produit par exemple lorsque $\forall r : sm_j^r = \{j\}$ et $sm_i^r = \{i, j\}$, où sm_i^r et sm_j^r désignent respectivement les vues obtenus par p_i et p_j au cours de la ronde r. Ainsi, bien que p_i effectue une infinité d'opérations d'écriture dans les objets partagés, il ne parvient pas à communiquer avec p_j. Nous pouvons dresser un parallèle avec le modèle de défaillances par *omission*. Les défaillances par omission modélisent la perte des messages par une faute de l'expéditeur ou du destinataire. Un processus est donc susceptible d'exhiber deux types de comportements déviants : la panne franche et la défaillance par omission. Dans ce dernier cas, le processus exécute correctement le code qui lui a été assigné mais omet de recevoir des messages qui lui sont destinés ou d'émettre des informations qu'il est supposé envoyer. Ainsi, étant donnée une exécution E dans ce modèle de défaillances, les processus peuvent être divisés en trois ensembles disjoints :

- $Crash(E)$: Les processus qui s'arrêtent inopinément.
- $Omission(E)$: Les processus qui ne connaissent pas de panne franche mais qui omettent d'envoyer certains de leurs messages.
- $Correct(E)$: Les processus qui ne défaillent pas (ni par panne franche, ni par omission) .

De façon similaire nous souhaitons dans le modèle *IIS* définir l'ensemble des processus corrects comme l'ensemble des processus qui exécutent une infinité d'opérations write_snap() et qui peuvent toujours communiquer entre eux.

Pour définir de façon précise cet ensemble, nous introduisons les définitions suivantes. Une exécution E dans le modèle $IRIS(PR_{\mathcal{C}})$ est complètement décrite par la donnée de la suite $(sm_i^r)_{(r\in\mathbb{N})}$ pour chaque processus p_i, où $sm_i^r = \{i_1, \ldots, i_x\}$ est l'ensemble des index des processus dont les valeurs apparaissent dans le snapshot obtenu par p_i lors de la ronde r. Si p_i exécute un nombre fini d'opérations write_snap(), $sm_i = \emptyset$ à partir d'un certain rang. Ces processus défaillent par panne franche. Dans ce qui suit, nous considérons uniquement les processus qui exécutent une infinité d'opérations write_snap() dans E. Soient deux processus p_i et p_j. La relation $p_i \xrightarrow{s} p_j$ traduit le fait que p_i observe p_j une infinité de fois. Formellement,

$$p_i \xrightarrow{s} p_j \text{ si et seulement si } \{r : j \in sm_i^r\} \text{ est infini}$$

$\xrightarrow{s}$ est réflexive car on a toujours $i \in sm_i^r$. De même, puisque qu'à chaque ronde r les ensembles sm_i^r sont ordonnées par inclusion, $\forall p_i, p_j : p_i \xrightarrow{s} p_j \vee p_j \xrightarrow{s} p_i$. Nous notons $\overset{s}{\rightsquigarrow}$ la fermeture transitive de la relation $\xrightarrow{s}$. L'ensemble des processus $obs(p_i)$ observés directement ou indirectement par p_i infiniment souvent est alors :

$$obs(p_i) = \{j : p_i \overset{s}{\rightsquigarrow} p_j\}$$

$\forall p_i : obs(p_i) \neq \emptyset$ car $\xrightarrow{s}$ est réflexive. De plus, $\forall p_i, p_j : obs(p_i) \subseteq obs(p_j) \vee obs(p_j) \subseteq obs(p_i)$ car $\forall p_i, p_j : p_i \xrightarrow{s} p_j \vee p_j \xrightarrow{s} p_i$. Par conséquent, il existe un plus petit ensemble obs_{min} pour l'inclusion. Cet ensemble caractérise l'ensemble des processus corrects dans E. En résumé :

- $Crash(E)$: l'ensemble des processus qui effectuent un nombre fini d'opérations write_snap() ;
- $Correct(E)$: $\min_{\subseteq}\{obs(p_i) : p_i \notin Crash(E)\}$;
- $Omission(E)$: $\{p_i : p_i \notin (Crash(E) \cup Correct(E))\}$.

La propriété suivante qui se déduit directement des définitions ci-dessus sera utilisée dans la suite.

Propriété 4.1 $\forall p_i \in Correct(E), \forall p_j \notin Correct(E) : \exists r$ *tel que* $\forall r' \geq r : j \notin sm_i^{r'}$.

4.1.2 Identifier les plus petits snapshots : get_smin()

Pour un objet $IS[r]$, les snapshots immédiats retournés sm_i^r sont ordonnés par inclusion. Les données contenues dans plus petit d'entre eux (noté $smin^r$) sont observées par tous les processus. Un tel snapshot représente donc le « savoir commun » minimal de la ronde r. Plusieurs algorithmes développés dans la suite reposent sur la capacité d'identifier un tel savoir commun. Plus généralement, pour un ensemble $X \subseteq \Pi$ donné, on cherche à identifier le plus petit snapshot parmi les snapshots retournés par l'objet $IS[r]$ pour les processus appartenant à X. Ce plus petit snapshot est noté $smin^r_{|X}$, i.e.,

$$smin^r_{|X} = \min_{\subseteq}\{sm_i^r : i \in X\}$$

A partir du seul snapshot sm_i^r obtenu lors de la ronde r, p_i ne peut en général pas déterminer $smin^r$ (le seul cas dans lequel p_i identifie $smin^r$ est lorsque $sm_i^r = \{i\}$). Cependant, en collectant les snapshots obtenus par les autres processus au cours des n dernières rondes, p_i est en mesure de déduire le plus petit snapshot $smin^\rho$ associé à l'une des n dernières rondes ($n - r \leq \rho < r$).

Ainsi, on se donne un suite de $n+1$ objets IS consécutifs et un ensemble $X \subseteq \Pi$. Pour simplifier, ces objets sont numérotés de 1 à $n+1$. Pour chaque processus p_i, i appartenant à X, le but consiste à identifier l'un des plus petits snapshots retournés par les appels $IS[r]$.write_snap(), $1 \leq r \leq n$ des processus de X.

Définition 4.1 (Spécification de la primitive get_smin(X))
Soit $IS[r], \ldots, IS[r+n]$ *une suite de* $n+1$ *objets snapshots immédiats consécutifs tels que les processus invoquent successivement* write_snap() *sur chacun d'entre eux. Chaque processus* p_i *doit produire en sortie un ensemble* $smin_i$ *tel que*

- $i \in X \Rightarrow smin_i \neq \emptyset$ *et,*
- $smin_i \neq \emptyset \Rightarrow (\exists \rho : r \leq \rho \leq r+n \wedge smin_i = smin^\rho_{|X})$.

Autrement dit, la spécification requiert que pour tout processus $p_i, i \in X$, $smin_i$ contienne un plus petit snapshot $smin^r_{|X}$ « récent », c'est à dire retourné lors d'une ronde à distance au plus $\delta \leq n$ de la ronde correspondant au dernier objet IS accédé. Deux processus p_i et p_j qui invoquent get_smin(X) n'obtiennent pas nécessairement le même plus petit snapshot $smin^\rho_{|X}$.

Algorithme Une implémentation de la primitive get_smin() est décrite dans la Figure 4.5. Chaque processus p_i maintient deux variables locales h_i et $smin_i$. La variable $smin_i$ est destinée à contenir le snasphot renvoyé par p_i comme résultat de l'appel get_smin(X).

```
operation : get_smin(X)
(01) h_i[1 : n+1] ← [∅, ..., ∅] ; smin_i ← ∅ ;
(02) for ℓ from 1 to n+1 do
(03)     sm_i ← IS[r+ℓ].write_snap(< i, h_i >) ;
(04)     foreach m ∈ {1, ..., ℓ−1} do h_i[m] ← ⋃{h_j[m] :< j, h_j >∈ sm_i} endfor ;
(05)     let ids(sm_i) = {j : ∃ < j, _ >∈ sm_i} ;
(06)     h_i[ℓ] ←< i, ids(sm_i) >
(07) endfor ;
(08) for m from 1 to n do
(09)     if ∃sm s.t. ∀j ∈ sm ∩ X :< j, sm >∈ h_i[m] then smin_i ← sm endif
(10) endfor ;
(11) return(smin_i)
```

FIG. 4.5 – Déterminer un plus petit snapshot (code pour p_i)

Le tableau local h_i représente la connaissance de p_i sur les snapshots retournés aux rondes $r+1, \ldots, r+n+1$. Chaque entrée $h_i[m]$ contient des éléments de la forme $< j, \{id_1, \ldots, id_x\} >$ (lignes 05-06). Ceci signifie que lors de la ronde $r+m$, l'ensemble des index qui apparaissent dans le snapshot obtenu par p_j est $\{id_1, \ldots, id_x\}$.

A chaque tour de la première boucle **for** (lignes 02-07), p_i enrichit sa connaissance sur les rondes passées en mettant à jour son histoire h_i avec les histoires h_j des processus p_j qu'il observe dans sm_i (ligne 04). Enfin, pour une ronde r, un snapshot s est le plus petit snapshot retourné par les opérations $IS[r]$.write_snap() des processus de X si et seulement si $\forall j \in s \cap X : s = sm_j^r$, i.e., si pour tous les processus de X dont l'identité apparaît dans s le résultat de l'opération $IS[r]$.write_snap() est s. Dans la seconde boucle **for** (lignes 08-10), p_i utilise cette caractérisation pour identifier l'un des $smin^\rho_{|X}$, $r+1 \leq \rho \leq r+n$.

Le Lemme 4.1 montre que dans le modèle *IIS*, il suffit de $n+1$ rondes pour un processus p_i, $i \in X$ pour identifier l'un des $smin^\rho_{|X}$, quelque soit le cardinal de X. Plus précisément, ce lemme montre que lorsque les ensembles $smin_{|X} \cap X$ sont inclus dans un même ensemble de cardinal k pendant $k+1$ rondes, chaque processus appartenant à X est capable d'identifier l'un d'entre eux à l'issue de ces $k+1$ rondes.

La démonstration du Lemme 4.1 repose uniquement les propriétés du modèle *IIS*. Il restera donc valide dans les restrictions $IRIS(PR_{\mathcal{C}})$ introduites dans le paragraphe 4.2.

Lemme 4.1 *Soit r un numéro de ronde, $X \subseteq \Pi$ et $i \in X$. $smin_i$ dénote le résultat de l'opération get_smin(X) pour le processus p_i. $\forall 1 \leq k \leq n$:*

$$\Big| \bigcup_{(r+n+1)-k \,\leq\, \rho \,\leq\, r+n+1} smin^\rho_{|X} \cap X \Big| \leq k \Rightarrow \exists \rho \in \{(r+n+1)-k, \ldots, r+n+1\} : smin_i = smin^\rho_{|X}$$

Cette équation peut être reformulée de la façon suivante. Notons $W(X,r)$ les identités des processus qui apparaissent à la fois dans X et dans $smin^r_{|X}$. Plus l'union des ces ensembles sur plusieurs rondes consécutives est petite, moins le nombre de rondes nécessaire à l'identification d'un plus petit snapshot est important. Par exemple, prenons $X = \Pi$ et supposons que $\cup W(r,X) = \{1\}$. Considérons deux rondes consécutives ρ et $\rho+1$. A l'issue de la ronde $\rho+1$, tous les processus connaissent l'histoire de p_1 car $\forall i \neq 1 : sm_1^{\rho+1} \subsetneq sm_i^{\rho+1}$. Cette histoire indique que p_1 a obtenu le plus petit snapshot de la ronde ρ car $|sm_1^\rho| = 1$.

Démonstration Nous démontrons le lemme par récurrence. Soit $HR(k)$ la propriété énoncée dans le lemme :

$$\Big|\bigcup_{(r+n+1)-k \,\leq\, \rho \,\leq\, r+n+1} smin^\rho_{|X} \cap X\Big| \leq k \Rightarrow \exists \rho \in \{(r+n+1)-k, \ldots, r+n+1\} : smin_i = smin^\rho_{|X}$$

- $HR(1)$. Il existe j tel que $smin^{r+n}_{|X} \cap X = smin^{r+1+n}_{|X} \cap X = \{j\}$. A la fin de la ronde $r+n$, $h_j[n] = \{< j, \{j, id_{j1}, \ldots, id_{j\alpha}\} >\}$ avec $id_{j1}, \ldots, id_{j\alpha} \notin X$et p_j écrit h_j lors de la ronde $r+n+1$. Lors de cette dernière ronde, $sm_j \subseteq sm_i$ car la vue de p_j est la plus petite parmi les vues des processus de X. Par conséquent, $< j, \{j, id_{j1}, \ldots, id_{j\alpha}\} > \in h_i[n]$ lorsque p_i exécute la deuxième boucle **for**, d'où $smin_i = smin^{r+n}_{|X}$.
- $HR(k) \Rightarrow HR(k+1)$.
  - $\forall j \in smin^{(r+n+)-(k+1)}_{|X} \cap X : \exists \rho \in \{(r+n+1)-k, \ldots, r+n+1\}$ tel que $j \in smin^\rho_{|X}$. Autrement dit, chaque identité j de la plus petite vue de la ronde $(r+n+1)-(k+1)$ apparaît à nouveau dans la plus petite vue d'une des rondes $(r+n+1)-k, \ldots, r+n+1$. Lors d'une telle ronde, p_i observe l'histoire h_j de p_j. p_i améliore alors sa connaissance des vues de la ronde $(r+n+1)-(k+1)$ en incluant $< j, sm_j^{(r+n+1)-(k+1)} >$ dans $h_i[(r+n+1)-(k+1)]$. Par conséquent, à l'issue de la ronde $r+n+1$, $h_i[(r+n+1)-(k+1)]$ contient $< j, sm_j^{(r+n+1)-(k+1)} >$ pour chaque $j \in smin^{(r+n+1)-(k+1)} \cap X$. Ainsi, p_i identifie la plus petite vue $smin^{(r+n+1)-(k+1)}_{|X}$ de la ronde $(r+n+1)-(k+1)$.
  - Le deuxième cas est la négation du cas précédent. $\exists j \in smin^{(r+n+)-(k+1)}_{|X} \cap X$ tel que $\forall \rho \in \{(r+n+1)-k, \ldots, r+n+1\} : j \notin smin^\rho_{|X}$. Nous avons donc $j \notin \bigcup_{(r+n+1)-(k) \,\leq\, \rho \,\leq\, r+n+1} smin^\rho_{|X}$, d'où $|\bigcup_{(r+n+1)-(k) \,\leq\, \rho \,\leq\, r+n+1} smin^\rho_{|X} \cap X| \leq k$. Par conséquent, $HR(k)$ s'applique.

$\square_{Lemme\ 4.1}$

4.2 Restriction du domaine de la lutte : $IRIS(PR_{\mathcal{C}})$

Dans ce paragraphe, nous montrons comment définir un modèle Snapshot Immédiat Restreint Itéré ($IRIS(PR_{\mathcal{C}})$) associé aux détecteurs $\mathcal{C}$ appartenant aux familles

$(\Omega^z)_{1\leq z\leq n}, (\diamond\mathcal{S}_x)_{1\leq x\leq n}$ et $(\diamond\psi^y)_{0\leq y\leq n-1}$. Le modèle $IRIS(PR_{\mathcal{C}})$ est induit par l'ensemble des exécutions du modèle *IIS* qui satisfont une certaine propriété $PR_{\mathcal{C}}$. Tous ces détecteurs sont « inéluctables » : dans chaque exécution leurs sorties vérifient une certaine propriété à partir d'un certain temps. Ceci implique que dans toute exécution finie, toute sortie arbitraire est légale par rapport à la spécification de ces détecteurs, quelque soit le motif de défaillances.

Les propriétés $PR_{\mathcal{C}}$ sont donc définies sur l'ensemble des exécutions infinies du modèle *IIS*. Elles sont de la forme : « dans chaque exécution infinie, il existe une ronde R à partir de laquelle les snapshots retournés vérifient une certaine propriété P ». D'un autre point de vue, une contrainte sur les snapshots retournés par les accès à un objet *IS* peut être exprimée comme des contraintes d'ordonnancement des processus (i.e., une contrainte sur la répartition des processus sur les « marches », cf. Figure 4.3).

Pour illustrer l'approche et essayer de donner l'intuition qui sous-tend les modèles $IRIS(PR_{\mathcal{C}})$, considérons le détecteur Ω. Rappelons qu'un tel détecteur fournit à chaque processus l'identité d'un leader tel que, à partir d'un certain temps, tous les processus perçoivent toujours la même identité, et cette identité est celle d'un processus correct. Comment les processus tirent profit de ce détecteur pour résoudre certaines tâches de décision ? Typiquement, chaque processus attend de recevoir des informations en provenance du processus qu'il perçoit comme leader avant d'exécuter une phase de vérification.

Ainsi, la propriété PR_{Ω} doit assurer que dans chaque exécution, il existe un processus p_ℓ et une ronde R à partir de laquelle chaque snapshot immédiat sm_i^r $(r \geq R)$ contient la valeur écrite par p_ℓ lors de cette ronde. Autrement dit, $\forall r \geq R, \forall i : sm_\ell^r \subseteq sm_i^r$. Il s'avère qu'il est possible à l'aide du détecteur Ω de contraindre davantage les exécutions : les exécutions qui satisfont la propriété PR_{Ω} sont telles que $\exists R, \exists \ell : \forall r \geq R : sm_\ell^r \subset sm_i^r$. Autrement dit, dans chaque exécution du modèle $IRIS(PR_{\Omega})$, il existe un processus p_ℓ et une ronde R à partir de laquelle le processus p_ℓ est toujours ordonnancé avant les autres processus, i.e., à partir de cette ronde les opérations write_snap() de p_ℓ sont set-linéarisées seules et avant toute autre opération.

Réciproquement, il est aisé d'implémenter un détecteur Ω dans le modèle $IRIS(PR_{\Omega})$: lors de chaque ronde, chaque processus p_i écrit son index i ; régulièrement, les processus identifient un plus petit snapshot récent (par exemple à l'aide de la primitive get_smin() présentée dans le paragraphe 4.1.2). A partir d'une certaine ronde, chacun de ces snapshots sera le même singleton $\{\ell\}$. Les Figures 4.6, 4.7 et 4.8, illustrent comment le modèle $IRIS(PR_{\Omega})$ induit une restriction du modèle *IIS*. Dans cet exemple, la propriété PR_{Ω} est satisfaite à partir de la deuxième ronde pour le processus p_1. Les exécutions du modèle *IIS* interdites par PR_{Ω} sont grisées.

4.2.1 Définition des modèles $IRIS(PR_{\mathcal{C}})$

Nous notons restricted_w_snap() les opérations qui satisfont les propriétés d'auto inclusion, de comparaison et d'immédiateté (définition 1.3.3) ainsi que la propriété additionnelle $PR_{\mathcal{C}}$. Les spécifications des propriétés PR_{Ω^z}, $PR_{\diamond\mathcal{S}_x}$ et $PR_{\diamond\psi^y}$ supposent que lors de chaque ronde, l'index i d'un processus p_i qui invoque restricted_w_snap() fait

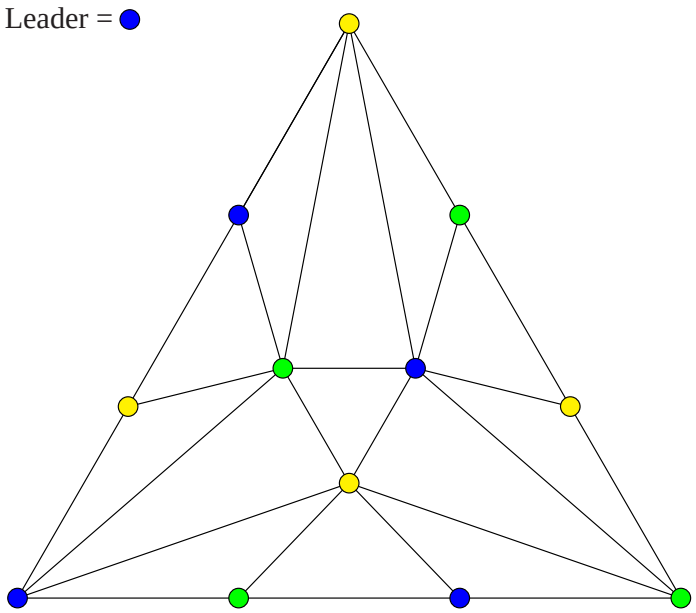

FIG. 4.6 – $IRIS(PR_\Omega)$, $R = 2$, $p_\ell = p_1$: ronde 1

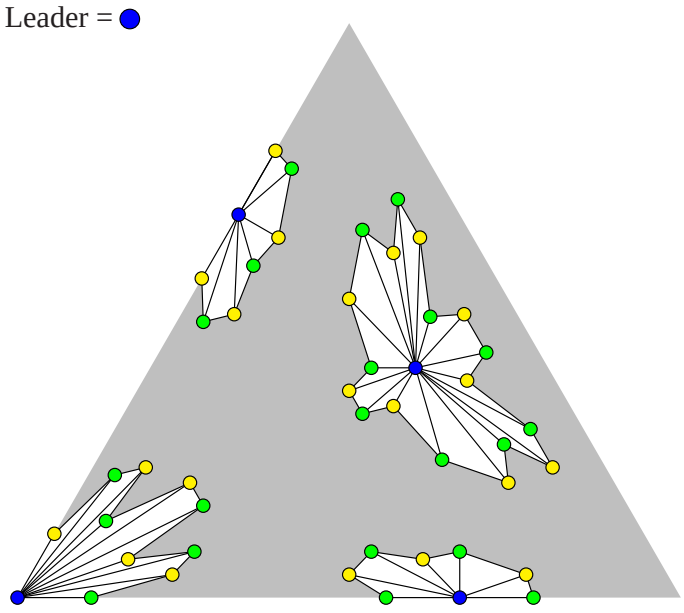

FIG. 4.7 – $IRIS(PR_\Omega)$, $R = 2$, $p_\ell = p_1$: ronde 2

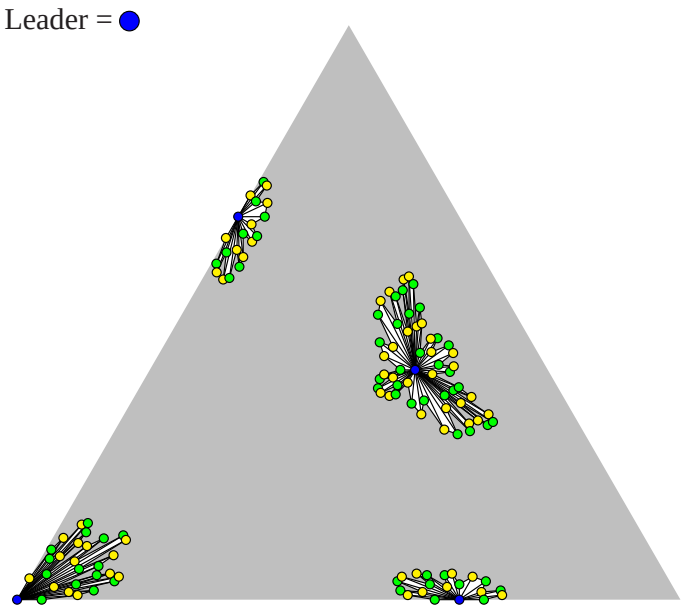

FIG. 4.8 – $IRIS(PR_{\Omega})$, $R = 2$, $p_\ell = p_1$: ronde 3

partie des données qu'il écrit. Si d'autres valeurs qui dépendent de l'algorithme qui est exécuté dans le modèle $IRIS(PR_{\mathcal{C}})$ correspondant sont aussi écrites, elles sont ignorées dans les spécifications qui suivent.

sm_j^r désignent le snapshot (parfois appelée vue) obtenu par le processus p_j comme réponse à l'invocation $IS[r]$.restricted_w_snap() (). Les propriétés $PR_{\mathcal{C}}$ n'ont de sens que dans des exécutions infinies : nous supposons donc que chaque processus exécute une infinité de rondes (sauf s'il défaille). Par convention, $sm_j^r = \emptyset$ si p_j n'exécute pas la ronde r, c'est à dire si p_j tombe en panne avant d'invoquer $IS[r]$.restricted_w_snap().

$IRIS(PR_{\diamond\mathcal{S}_x})$ La propriété $PR_{\diamond\mathcal{S}_x}$ est définie de la façon suivante (Q est un ensemble d'identités de processus) :

$$\begin{array}{rcl} PR_{\diamond\mathcal{S}_x} & \equiv & \exists Q, \ell : |Q| \geq x \ \wedge \ \ell \in Q : \\ & & \exists r : \forall r' \geq r : (sm_\ell^{r'} \neq \emptyset) \ \wedge \ \big(i \in Q \setminus \{\ell\} \ \Rightarrow \ (sm_i^{r'} = \emptyset \ \vee \ sm_\ell^{r'} \subsetneq sm_i^{r'})\big) \end{array}$$

Autrement dit, dans chaque exécution de ce modèle, il existe un ensemble Q d'au moins x processus et un processus p_ℓ tels que, à partir d'une certaine ronde chaque vue des processus de $Q - \{\ell\}$ contient strictement la vue obtenue par le processus p_ℓ. Du point de vue ordonnancement (ou « répartition sur les marches ») , p_ℓ est, à partir de cette ronde toujours ordonnancé avant les processus de $Q - \{\ell\}$.

IRIS$(PR_{\diamond\psi^y})$ Rappelons que le nombre de processus qui défaillent dans une exécution est noté f. Un détecteur de défaillances $\diamond\psi^y$ donne une estimation du nombre de processus corrects. Dans le modèle *IIS*, p_i considère le processus p_j correct s'il apparaît une infinité de fois dans sa vue. C'est à dire qu'il existe une infinité de rondes r telle que $j \in sm_i^r$. La propriété $PR_{\diamond\psi^y}$ exprime cette idée de la façon suivante : p_i obtient une infinité de vues sm_i qui contiennent tous les processus corrects lorsque $f > n - y$. Lorsque le nombre de défaillances ne dépasse pas le seuil $n-y$, p_i obtient une infinité de vues sm_i qui contiennent les identités d'au moins y processus corrects. Formellement,

$$\begin{aligned} PR_{\diamond\psi^y} \quad \equiv \quad & \exists r : \forall r' \geq r : \\ & \big((i-1 = r' \mod n) \ \wedge \ (sm_i^{r'} \neq \emptyset)\big) \Rightarrow |sm_i^{r'}| \geq n - \max(n-1-y, f) \end{aligned}$$

IRIS(PR_{Ω^z}) La spécification de la propriété PR_{Ω^z} est (L est un ensemble de d'identités de processus) :

$$\begin{aligned} PR_{\Omega^z} \quad \equiv \quad & \exists L : |L| \leq z \\ & \exists r : \forall r' \geq r : \exists i \in L : (sm_i^{r'} \neq \emptyset) \ \wedge \ (sm_i^{r'} \subseteq L) \end{aligned}$$

Ou, de manière équivalente ($smin^r$ est le plus petit snapshot retourné lors de la ronde r) :

$$\begin{aligned} PR_{\Omega^z} \quad \equiv \quad & \exists L : |L| \leq z \\ & \exists r : \forall r' \geq r : smin^{r'} \subseteq L \end{aligned}$$

Autrement dit, il existe une ronde r et un ensemble de processus L de cardinal au plus z tel que à partir de la ronde r chaque processus observe toujours dans sa vue au moins une identité d'un processus appartenant à L. Ceci implique qu'à chaque ronde $r' \geq r$, le (ou les) processus qui effectue(nt) la (ou les) opérations set-linéarisée(s) en premier appartiennent toujours à L.

4.2.2 Constructions de *IRIS*$(PR_{\mathcal{C}})$ dans le modèle lire/écrire

Dans ce paragraphe, nous présentons des constructions qui simulent les modèle *IRIS*$(PR_{\mathcal{C}})$, $\mathcal{C} \in \{\diamond\mathcal{S}_x, \diamond\psi^y, \Omega^z]\}$ dans le modèle à mémoire partagée équipé d'un détecteur de défaillance de la classe C. Le nombre de processus défaillants est arbitraire. Nous supposons seulement qu'il existe toujours au moins un processus correct dans toute exécution (i.e., $t = n-1$). La notation $\mathcal{SM}_{n,n-1}[\mathcal{C}]$ désigne ce modèle.

Les constructions implémentent l'opération $IS[r]$.restricted_w_snap() où $IS[1\ldots+\infty)$ définit la séquence infinie d'objets snapshot immédiat du modèle *IRIS*$(PR_{\mathcal{C}})$ que l'on souhaite construire. Chacune des opérations restricted_w_snap() doit respecter les propriétés d'auto-inclusion, de comparaison et d'immédiateté. De plus, la propriété $PR_{\mathcal{C}}$ doit être satisfaite dans toute suite infinie d'opérations $IS[1]$.restricted_w_snap(), $IS[2]$.restricted_w_snap(), ...

Objet snapshot immédiat observable Les constructions reposent sur l'utilisation d'objets snapshot immédiat *observables* notés $R[1], R[2], \ldots$ Un objet $R[r]$ supporte deux opérations write_snap() et snap(). Une invocation $R[r]$.write_snap() retourne une vue qui satisfait les propriétés d'auto-inclusion, de comparaison et d'immédiateté. Les vues retournées par les invocations $R[r]$.snap() satisfont la propriété de comparaison. De plus, pour un objet $R[r]$ donné, nous contraignons le motif des accès : chaque processus a le droit à au plus une invocation de la primitive write_snap() et cet appel est la dernière opération effectuée par le processus sur l'objet $R[r]$. Par contre, le nombre d'appels snap() n'est pas limité.

```
init  LEVEL[1..n] ← [n+1, ..., n+1];
      VAL[1..n] ← [⊥, ..., ⊥];

operation R[r].write_snap(v_i)
(1)   VAL[i] ← v_i;
(2)   repeat LEVEL[i] ← LEVEL[i] − 1;
(3)        foreach j ∈ {1, ..., n} do level_i[j] ← LEVEL[j] enddo;
(4)        view_i ← {j : level_i[j] ≤ LEVEL[i]};
(5)   until (|view_i| = LEVEL[i]) endrepeat;
(6)   S_i ← {(j, VAL[j]) : j ∈ view_i}; return(S_i)

operation R[r].snap()
(7)   ℓ_i ← n + 1;
(8)   repeat ℓ_i ← ℓ_i − 1;
(9)        foreach j ∈ {1, ..., n} do level_i[j] ← LEVEL[j] enddo;
(10)       view_i ← {j : level_i[j] ≤ ℓ_i};
(11)  until (|view_i|ℓ_i) endrepeat;
(12)  S_i ← {(j, VAL[j]) : j ∈ view_i}; return(S_i)
```

FIG. 4.9 – Un objet snapshot immédiat « observable » (code pour p_i)

Le code de la Figure 4.9 implémente un objet R en mémoire partagée de façon sans attente. Le code de l'opération write_snap() est l'algorithme de l'escalier de Borowsky et Gafni [22]. Pour obtenir un snapshot, un processus p_i parcourt en descendant les niveaux jusqu'à trouver un niveau « plein ». Un niveau ℓ est plein si le cardinal de l'ensemble des processus situés sur les niveaux inférieurs ou égaux à ℓ est au moins égal à ℓ (c'est à dire $|\{j : LEVEL[j] \leq \ell\}| \geq \ell$). Il est aisé de vérifier que les vues obtenues en réponse aux invocations snap() ou write_snap() sont ordonnées par inclusion. De même, puisque les opérations snap() ne modifient les registres, la correction de l'algorithme originel [22] implique les vues retournées par les opérations write_snap() satisfont en sus les propriété d'auto inclusion et d'immédiateté.

Soit m_i une vue obtenue par p_i en réponse à un appel snap() et sm_i le snapshot retourné par l'appel write_snap(). La propriété suivante découle directement du code.

Propriété 4.2 $j \in m_i \Rightarrow sm_j \subsetneq sm_i$

Forme générale des simulations Les constructions suivent le patron suivant :

operation $IS[r]$.restricted_w_snap$(< i, v_i >)$:
initialisation
repeat $m_i \leftarrow R[r]$.snap() ;
calculs
until P **endrepeat** ;
$sm_i \leftarrow R[r]$.write_snap$(< i, v_i >)$;
return (sm_i)

Les étapes « initialisation » et « calculs » à l'intérieur de la boucle ainsi que le prédicat P dépendent du modèle $IRIS(PR_{\mathcal{C}})$ simulé et du détecteur $\mathcal{C}$ dont le modèle à mémoire partagée est équipé. La dernière opération qui affecte la mémoire partagée est toujours $sm_i \leftarrow R[r]$.write_snap$(< i, v_i >)$. L'ensemble sm_i produit est le résultat de cette invocation. Ainsi, les propriétés d'auto-inclusion, de comparaison et d'immédiateté sont donc satisfaites car ces ensembles vues sont obtenues par l'intermédiaire de l'objet $R[r]$ qui garantit ces propriétés (c'est à dire que sm_i est un snapshot immédiat valide). Pour chacune des constructions qui suivent, nous devons donc seulement démontrer que la propriété $PR_{\mathcal{C}}$ est satisfaite ainsi que la vivacité de la construction : pour chaque processus correct, le prédicat P est inéluctablement satisfait.

4.2.3 Simulation de $IRIS(PR_{\diamond\mathcal{S}_x})$ dans le modèle $\mathcal{SM}_{n,n-1}[\diamond\mathcal{S}_x]$

Un algorithme qui simule les opérations du modèle $IRIS(PR_{\diamond\mathcal{S}_x})$ est décrit dans la Figure 4.10. Nous considérons ici la version « représentant » de la définition de $\diamond\mathcal{S}_x$ (voir la définition 3.8, paragraphe 3.4 , page 105). Chaque processus p_i met régulièrement à jour deux variables locales m_i et rp_i. m_i est la vue de p_i de l'état de la mémoire partagée tandis que rp_i contient le représentant courant de p_i indiqué par le détecteur de défaillances.

operation $IS[r]$.restricted_w_snap$(< i, v_i >)$:
(1) **repeat** $m_i \leftarrow R[r]$.snap() ;
(2) $rp_i \leftarrow \text{REPR}_i$
(3) **until** $(< rp_i, - > \in m_i) \ \vee \ (rp_i = i)$ **endrepeat** ;
(4) $sm_i \leftarrow R[r]$.write_snap$(< i, v_i >)$;
(5) return (sm_i)

FIG. 4.10 – De $\mathcal{SM}_{n,n-1}[\diamond\mathcal{S}_x]$ vers $IRIS(PR_{\diamond\mathcal{S}_x})$ (code pour p_i)

Proposition 4.1 *L'algorithme de la Figure 4.10 simule le modèle $IRIS(PR_{\diamond\mathcal{S}_x})$ dans le modèle $\mathcal{SM}_{n,n-1}[\diamond\mathcal{S}_x]$.*

Démonstration Nous démontrons dans un premier temps que les opérations restricted_w_snap() terminent. Nous démontrons ensuite que la propriété $PR_{\diamond\mathcal{S}_x}$ est satisfaite. Les propriétés d'auto-inclusion, de comparaison et d'immédiateté découlent du fait que la vue retournée sm_i est obtenue par l'intermédiaire d'un appel $R[r]$.write_snap$(< i, v_i >)$.

– Terminaison $\forall r$: pour un processus correct, l'appel $IS[r]$.restricted_w_snap$(< i, v_i >)$ termine.

La démonstration repose sur les deux observations suivantes qui découlent immédiatement du code :

1. Soit p_i un processus correct. Si au bout d'un certain temps, on a toujours $\text{REPR}_i = i$ alors p_i ne peut être bloqué indéfiniment dans la boucle **repeat**.
2. Au cours d'une ronde r, s'il existe un processus correct p_i qui invoque $R[r]$.write_snap() alors tous les processus corrects p_j dont le représentant REPR_j est au bout d'un certain temps toujours p_i ne peuvent être bloqués indéfiniment dans la boucle **repeat**.

Soit p_i un processus correct et r un numéro de ronde arbitraire. Par définition de la classe $\diamond\mathcal{S}_x$, il existe un instant à partir duquel le représentant de p_i ne change plus et est un processus correct. Nous notons p_ℓ ce représentant « final » de p_i (éventuellement, $\ell = i$). De plus, ce représentant est tel que, au bout d'un certain temps, $\text{REPR}_\ell = \ell$ est toujours vrai.
D'après la première observation ci-dessus, toutes les opérations $IS[1]$ .restricted_w_snap(), $IS[2]$ .restricted_w_snap(), ..., $IS[r]$ .restricted_w_snap() de p_ℓ terminent. En particulier, p_ℓ écrit dans l'objet partagé $R[r]$. Par conséquent, p_i ne peut être bloqué indéfiniment dans la boucle **repeat** de la ronde r car, au bout d'un certain temps tous les vues m_i (retournées par les appels $R[r]$.snap()) contiennent l'identité ℓ du représentant de p_i.

– $PR_{\diamond\mathcal{S}_x}$
D'après la spécification de la classe $\diamond\mathcal{S}_x$ il existe au bout d'un certain temps un ensemble de processus Q de cardinal au moins x qui ont le même représentant. Les autres processus $\notin Q$ ont pour représentant leur propre identité. Plus formellement, il existe un instant τ, un ensemble de processus Q et un processus correct $p_\ell \in Q$ tels que :
– $|Q| \geq x$;
– Pour $p_i \notin Q$: $\forall \tau' \geq \tau : \text{REPR}_i^{\tau'} = i$;
– Pour $p_i \in Q$: $\forall \tau' \geq \tau : \text{REPR}_i^{\tau'} = \ell$.
Soit r une ronde qui démarre après l'instant τ. C'est à dire que chaque invocation $IS[r]$.restricted_w_snap() démarre à un instant plus grand que τ.
Par définition, p_ℓ est un processus correct. Puisque nous avons montré que tous les appels $IS[r]$ .restricted_w_snap() effectués par les processus corrects terminent, $sm_\ell^r \neq \emptyset$. Soit $p_i \in Q - \{\ell\}$ un processus qui termine son appel $IS[r]$ .restricted_w_snap(). Cet appel démarre après l'instant τ. Donc, lors de l'exécution de cet appel, nous avons toujours $rp_i = \ell$. Ainsi, lorsque le prédicat de sortie de la boucle **repeat** est satisfait, $\ell \in m_i$. De plus, d'après les propriétés de l'objet $R[r]$, nous avons $sm_\ell^r \subseteq m_i \subseteq sm_i^r$. De plus, $i \in sm_i^r$ et $i \notin m_i$ d'où $sm_\ell^r \subsetneq sm_i^r$.

□*Proposition* 4.1

4.2.4 Simulation de $IRIS(PR_{\diamond\psi^y})$ dans le modèle $\mathcal{SM}_{n,n-1}[\diamond\psi^y]$

La construction (Figure 4.11) s'inspire d'un algorithme de ℓ-exclusion mutuelle [2]. Nous souhaitons que périodiquement chaque processus p_i obtienne une vue $|sm_i| \geq$

$\min(1+y, n-f)$. Pour cela, nous associons à chaque ronde un ordre sur les identités des processus. L'idée sous-jacente est d'autoriser chaque processus p_i à écrire lorsque les processus qui le précèdent dans l'ordre associé à la ronde courante ont préalablement écrit. Ainsi, la vue *sm* obtenue par le processus de rang le plus élevé contient tous les processus corrects.

Cependant, les processus ne disposent pas d'un moyen fiable de calculer exactement le nombre de processus corrects qui les précèdent. Le détecteur de défaillances de la classe $\diamond\psi^y$ fournit seulement une estimation du nombre de processus défaillants. Cette information suffit néanmoins pour calculer un minorant du nombre de processus correct. Nous verrons dans la démonstration que, de cette façon, le processus de plus haut rang calcule une vue qui contient au moins $\min(y+1, n-f)$ processus corrects.

La fonction déterministe order(r) retourne une permutation π_r des entiers $1, \ldots, n$ telle que $\pi_r(n) = i$ si $(i-1) = r \mod n$. Soit μ la permutation miroir ($\mu : (a_1, \ldots, a_n) \rightarrow (a_n, \ldots, a_1)$) et π_c la permutation circulaire qui effectue un décalage à droite ($\pi_c : (a_1, \ldots, a_n) \rightarrow (a_n, a_1 \ldots, a_{n-1})$). Par exemple, π_r peut être définie ainsi : $\pi_r = \mu.\pi_c^{r-1}$.

Description de l'algorithme et démonstration Lorsqu'il invoque $IS[r]$.restricted_w_snap(), un processus p_i calcule la séquence ($sequence_i$) des identités des processus associée à la ronde r (ligne 1), et détermine l'ensemble ($pred_i$) des processus qui le précédent dans cette séquence (ligne 2). Ensuite, p_i entre dans une boucle au cours de laquelle il calcule l'ensemble $seen_i$ des processus qui (1) ont écrit dans l'objet $R[r]$ et (2) le précèdent selon l'ordre $sequence_i$ (ligne 4). Il lit également l'estimation (nbc_i) du nombre de processus défaillant fournie par le détecteur. Ces instructions sont répétées jusqu'à ce que les processus de $pred_i$ perçus comme corrects aient écrit dans $R[r]$ (ligne 6). p_i estime le nombre de ces processus à au moins $(|pred_i| - nbc_i)$. Enfin, lorsque p_i sort de la boucle, il écrit à son tour dans l'objet $R[r]$ (ligne 7) et le snapshot immédiat retourné par cette opération est le résultat final de l'appel (ligne 8).

```
operation IS[r].restricted_w_snap(< i, v_i >) :
(1) sequence_i ← order(r) ;
(2) pred_i ← {j : j appears before i in sequence_i} ;
(3) repeat  m_i ← R[r].snap() ;
(4)         seen_i ← m_i ∩ pred_i ;
(5)         nbc_i ← NB_C_i
(6) until (|pred_i| − nbc_i ≤ |seen_i|) endrepeat ;
(7) sm_i ← R[r].write_snap(< i, v_i >) ;
(8) return (sm_i)
```

FIG. 4.11 – De $\mathcal{SM}_{n,n-1}[\diamond\psi^y]$ vers $IRIS(PR_{\diamond\psi^y})$ (code pour p_i)

Nous démontrons d'abord que les appels $IS[r]$.restricted_w_snap() effectués par les processus corrects terminent (Lemme 4.2). La proposition 4.1 établit la correction de la simulation. La démonstration donnée dans [39] considère le cas plus général du modèle $\mathcal{SM}_{n,t}[\diamond\psi^y]$.

Lemme 4.2 *$\forall r, \forall p_i$ processus correct : les appels $IS[r]$.restricted_w_snap() de p_i terminent.*

Démonstration Pour établir une contradiction, supposons qu'il existe des rondes au cours desquelles des processus corrects ne sortent jamais de la boucle **repeat**. Soit r le plus petit numéro de ronde pour laquelle il existe un processus bloqué. Nous notons B l'ensemble des processus corrects bloqués dans la boucle **repeat** de la ronde r. Ainsi, pour tout processus $p_i \in B$, le prédicat $(|pred_i| - nbc_i \leq |seen_i|)$ n'est jamais satisfait. Parmi les processus de B, soit p_s le processus de plus petit rang selon l'ordre order(r). Dans ce qui suit, nous montrons que l'invocation $IS[r]$.restricted_w_snap() de p_s termine : une contradiction.

Nous distinguons deux cas en fonction du cardinal de $pred_s$ (f est le nombre de processus défaillant dans l'exécution considérée).

- $|pred_s| \leq \max(n-1-y, f)$
  D'après la propriété de convergence inéluctable de la classe $\diamond\psi^y$, il existe un instant τ à partir duquel $nbc_s = \max(n-1-y, f)$, d'où $|pred_s| - nbc_s \leq 0$. Donc, après τ, le prédicat de sortie de la boucle est vérifié car $0 \geq |m_i \cap seen_i|$ pour p_s.
- $|pred_s| > \max(n-1-y, f)$
  Notons $faulty(S)$ l'ensemble des processus défaillants dans l'ensemble S. Nous avons $|faulty(S)| \leq |faulty(\{1, \ldots, n\})| = f \leq \max(n-1-y, f)$.
  Notons α le nombre de processus corrects dans l'ensemble $pred_s$. Nous avons $\alpha = |pred_s| - |faulty(pred_s)| \geq |pred_s| - \max(n-1-y, f)$. Ces α processus corrects ont un rang inférieur à p_s dans l'ordre order(r). De plus, par définition de p_s, tous les processus corrects dont le rang est inférieur à celui de p_s dans order(r) terminent leur appel $IS[r]$.restricted_w_snap().
  Par conséquent, tous les processus corrects de $pred_s$ écrivent dans l'objet $R[r]$. Il existe donc un instant τ_0 à partir duquel m_s contient les identités de tous ces processus (m_s est la vue des processus qui ont écrit dans l'objet $R[r]$ et est régulièrement mise à jour par p_s). D'autre part, la propriété de convergence inéluctable de la classe $\diamond\psi^y$ assure l'existence d'un instant τ_1 à partir duquel $nbc_s = \max(n-1-y, f)$. Donc, après $\tau = \max(\tau_0, \tau_1)$, $|seen_s| = |m_s \cap pred_s| \geq \alpha$ et $\alpha \geq |pred_s| - nbc_s$. Nous en concluons que p_s sort de la boucle **repeat**, d'où la terminaison de l'invocation $IS[r]$.restricted_w_snap().

□*Lemme 4.2*

Proposition 4.2 *L'algorithme de la Figure 4.11 simule le modèle IRIS($PR_{\diamond\psi^y}$) dans le modèle $\mathcal{SM}_{n,n-1}[\diamond\psi^y]$.*

Démonstration Le Lemme 4.2 établit la terminaison des opération restricted_w_snap() par les processus corrects. Ce qui suit montre que dans toute exécution infinie, la propriété $PR_{\diamond\psi^y}$ est satisfaite.

Soit R une ronde telle que $\forall r \geq R, \forall p_i$ processus correct : lors de l'exécution de $IS[r]$.restricted_w_snap() par p_i, on a en permanence $nbc_i = \max(n-1-y, f)$.

Soit $r \geq R$ un numéro de ronde qui satisfait $(i-1) = r \mod n$ et nous supposons que p_i obtient sm_i en retour de l'appel $IS[r]$.restricted_w_snap(). Nous montrons que $|sm_i| \geq n - \max(n-1-y, f)$.

1. Le rang de p_i selon l'ordre order(r) est n, d'où $|pred_i| = n-1$ au cours de cette ronde. Le prédicat de sortie est satisfait car l'appel termine, d'où $|seen_i| \geq |pred_i| - nbc_i = (n-1) - \max(n-1-y, f)$.
2. Soit m_i le dernier ensemble obtenu par p_i (ligne 3). Puisque $i \notin m_i$ et $|pred_i| = n-1$, $m_i \cap pred_i = m_i = seen_i$.
3. Finalement, on sait que, grâce à l'objet $R[r]$, $m_i \subsetneq sm_i$ (propriété 4.2).

En combinant les relations précédentes, nous obtenons $|sm_i| > |m_i| = |seen_i| \geq (n-1) - \max(n-1-y, f)$, d'où $|sm_i| \geq n - \max(n-1-y, f)$. $\square_{Proposition\ 4.2}$

4.2.5 Simulation de $IRIS(PR_{\Omega^z})$ dans le modèle $\mathcal{SM}_{n,n-1}[\Omega^z]$

La construction est décrite dans la Figure 4.12. Pour satisfaire la propriété PR_{Ω^z}, l'algorithme repose sur l'idée simple suivante. Les processus indiqués « leaders » par le détecteur écrivent les premiers. Les autres processus ne sont autorisés à écrire qu'à la condition d'observer une écriture d'un de ces leaders.

Plus précisément, lorsque p_i invoque $IS[r]$.restricted_w_snap(), il attend qu'un processus ait écrit dans l'objet $R[r]$ ou que son identité apparaisse dans la sortie LEADER$_i$ du détecteur de défaillances. Lorsque qu'au moins l'une de ces conditions est satisfaite, p_i écrit dans l'objet $R[r]$ en invoquant $R[r]$.write_snap(). Le snapshot sm_i retourné par cette opération est le résultat de l'invocation $IS[r]$.restricted_w_snap().

```
operation IS[r].restricted_w_snap(< i, v_i >) :
(1) repeat  m_i ← R[r].snap() ;
(2)         ld_i ← LEADER_i
(3) until (m_i ≠ ∅) ∨ (i ∈ ld_i) endrepeat ;
(4) sm_i ← R[r].write_snap(< i, v_i >) ;
(5) return (sm_i)
```

FIG. 4.12 – De $\mathcal{SM}_{n,n-1}[\Omega^z]$ vers $IRIS(PR_{\Omega^z})$ (code pour p_i)

Proposition 4.3 *L'algorithme de la Figure 4.12 simule le modèle $IRIS(PR_{\Omega^z})$ dans le modèle $\mathcal{SM}_{n,n-1}[\Omega^z]$.*

Démonstration La démonstration se divise en deux parties. La première partie montre que les opérations $IS[r]$.restricted_w_snap() effectuées par les processus corrects terminent. La seconde partie établit que les vues $(sm_i^r)_{i\in\Pi, r\geq 1}$ satisfont la propriété PR_{Ω^z}.

– Terminaison $\forall r$: pour un processus correct, l'appel $IS[r]$.restricted_w_snap($< i, v_i >$) termine.
Soit r une ronde arbitraire et p_i un processus correct. Nous supposons que tous les processus corrects invoquent $IS[r]$.restricted_w_snap(). D'après la spécification de

la classe Ω^z, il existe un instant à partir duquel la sortie LEADER_i du détecteur se stabilise et est la même pour chaque processus. Soit L la sortie stabilisée du détecteur. Nous savons également que L contient l'identité d'un processus correct p_x (il est possible que $x = i$).
p_x ne peut être indéfiniment bloqué dans la boucle **repeat** puisque, inéluctablement la condition $(x \in \text{LEADER}_x)$ est toujours vraie. Par conséquent, p_x écrit dans l'objet $R[r]$. À l'issue de cette écriture, toute opération $R[r].\mathsf{snap}()$ retourne un ensemble m_i non vide. Par conséquent, p_i ne peut être bloqué indéfiniment dans la boucle **repeat**.

– PR_{Ω^z}
Soit τ l'instant de stabilisation de la sortie du détecteur. Plus précisément, il existe un ensemble L qui contient l'identité d'un processus correct tel que $\forall \tau' \geq \tau, \forall i : \text{LEADER}_i^{\tau'} = L$. Considérons une ronde r qui démarre après τ (c'est à dire que tous les appels $IS[r].\mathsf{restricted\_w\_snap}()$ sont lancés après τ). Soit $smin^r$ le plus petit snapshot immédiat retourné par les appels $R[r].\mathsf{write\_snap}()$. Nous montrons que $sm_m^r \subseteq L$. Soit $p_i \notin L$. Lorsque p_i sort de la boucle **repeat**, $m_i \neq \emptyset$. D'après les propriétés de l'objet $R[r]$, $smin^r \subseteq m_i$ et $m_i \subsetneq sm_i$. Or, $i \in sm_i^r$. Par conséquent, $i \notin smin^r$.

$\square_{Proposition\ 4.3}$

4.3 Construction d'un détecteur de la classe $\mathcal{C}$ dans le modèle $IRIS(PR_{\mathcal{C}})$

Nous avons vu dans la partie précédent comment simuler le modèle $IRIS(PR_{\mathcal{C}})$ dans le modèle standard à registres atomiques muni d'un détecteur $\mathcal{C}$. Ce paragraphe étudie la transformation inverse : Étant donné le modèle $IRIS(PR_{\mathcal{C}})$, comment construire un détecteur de la classe $\mathcal{C}$?

4.3.1 $\diamond\mathcal{S}_x$ dans le modèle $IRIS(PR_{\diamond\mathcal{S}_x})$

Une construction simple est décrite dans la Figure 4.13. L'ensemble TRUSTED_i est régulièrement mis à jour avec sm_i, où sm_i est le résultat du dernier appel $\mathsf{restricted\_w\_snap}()$.

```
init r_i ← 0 ; TRUSTED_i ← Π

(1) repeat r_i ← r_i + 1 ;
(2)        sm_i ← IS[r].restricted_w_snap(i) ;
(3)        TRUSTED_i ← sm_i
(4) until false endrepeat
```

FIG. 4.13 – $\diamond\mathcal{S}_x$ dans $IRIS(PR_{\diamond\mathcal{S}_x})$ (code for p_i)

Proposition 4.4 *L'algorithme de la Figure 4.13 construit un détecteur de défaillances de la classe $\diamond\mathcal{S}_x$ dans le modèle $IRIS(PR_{\diamond\mathcal{S}_x})$.*

Démonstration Soit E une exécution infinie arbitraire dans le modèle $IRIS(PR_{\diamond\mathcal{S}_x})$. Nous devons établir qu'il existe un ensemble X et un processus correct p_ℓ qui satisfont inéluctablement les propriétés suivantes :

1. $|X| \geq x$ et $\forall p_i \in X \cap \mathit{Correct}(E) : \ell \in \text{TRUSTED}_i$;

De plus, à partir d'un certain instant, l'ensemble TRUSTED_i pour un processus correct p_i ne contient plus que des identités de processus corrects :

2. $\forall p_i \in \mathit{Correct}(E) : \text{TRUSTED}_i \subseteq \mathit{Correct}(E)$.

Soit R une ronde telle que :
- La propriété $PR_{\diamond\mathcal{S}_x}$ est satisfaite $\forall r \geq R$;
- $\forall p_i \in \mathit{Correct}(E), \forall r \geq R : sm_i^r \subseteq \mathit{Correct}(E)$.

La propriété 4.1 relative à la notion de processus corrects dans les modèles itérés et le fait que l'exécution se déroule dans le modèle $IRIS(PR_{\diamond\mathcal{S}_x})$ assurent l'existence de R. D'où, après R, l'ensemble TRUSTED_i pour chaque processus correct p_i est toujours un sous-ensemble des processus corrects. La condition 2 est vérifiée.

Soient X ($|X| \geq x$) et p_ℓ l'ensemble et le processus de la propriété $PR_{\diamond\mathcal{S}_x}$. Si X inclut l'identité d'un processus correct p_i alors, par définition de $\mathit{Correct}(E)$, p_ℓ est aussi correct car ℓ apparaît toujours dans les vues $(sm_i^r)_{r\geq R}$. De plus, d'après le code, $\ell \in \text{TRUSTED}_i$ après la ronde R. Enfin, si X ne contient que des processus défaillants, n'importe quel processus correct convient pour le choix de ℓ. En effet, grâce à la propriété d'auto-inclusion, on a toujours $i \in \text{TRUSTED}_i$. $\square_{\mathit{Proposition}\ 4.4}$

4.3.2 $\diamond\psi^y$ dans le modèle $IRIS(PR_{\diamond\psi^y})$

L'algorithme décrit en Figure 4.14 implémente un détecteur de la classe $\diamond\psi^y$ dans le modèle $IRIS(PR_{\diamond\psi^y})$. Il partage la simplicité de l'algorithme précédent. p_i utilise directement la dernière vue obtenue sm_i pour mettre à jour la variable NB_C_i.

```
init : r_i ← 0 ; NB_C_i ← (n − 1 − y)

(1) repeat r_i ← r_i + 1 ;
(2)        sm_i ← IS[r].restricted_w_snap(i) ;
(3)        if (i − 1) = (r_i mod n) then NB_C_i ← max(n − 1 − y, n − |sm_i|) end if
(4) until false endrepeat
```

FIG. 4.14 – $\diamond\psi^y$ dans $IRIS(PR_{\diamond\psi^y})$

Il suit immédiatement du code et de la propriété $PR_{\diamond\psi^y}$ que NB_C_i converge vers $\max(n-1-y, f)$.

Proposition 4.5 *L'algorithme de la Figure 4.14 construit un détecteur de défaillances de la classe $\diamond\psi^y$ dans le modèle $IRIS(PR_{\diamond\psi^y})$.*

Remarque Nombre de processus corrects.
Soit E une exécution infinie arbitraire. Dans la définition de $PR_{\diamond\psi^y}$, f désigne le nombre de processus qui tombent en panne, i.e., qui effectuent un nombre fini d'opérations restricted_w_snap(). On a vu que les processus qui effectuent une infinité d'opérations restricted_w_snap() se répartissent en deux ensembles $Omission(E)$ et $Correct(E)$. Quel est le cardinal de l'ensemble $Correct(E)$?

Soit p_i un processus correct et r une ronde telle que
- $(i-1) = r \mod n$;
- $r \geq R$, la ronde à partir de laquelle la propriété $PR_{\diamond\psi^y}$ est satisfaire ;
- $sm_i^r \subseteq Correct(E)$ (il existe une ronde R' à partir de laquelle ceci est toujours vérifié d'après la propriété 4.1).

D'où $|Correct(E)| \geq |sm_i^r| \geq \min(y+1, n-f)$.
- $n - f \leq y + 1$. Dans ce cas, le nombre de processus corrects est exactement $n - f$.
- $n - f > y + 1$. On a alors $n - f \geq |Correct(E)| \geq y + 1$, d'où $n - f - (y + 1) \geq |Omission(E)| \geq 0$. Ceci est cohérent avec la sortie du détecteur qui est une estimation du nombre de processus défaillant par panne franche ou par omission.

4.3.3 Ω^z dans le modèle $IRIS(PR_{\Omega^z})$

Dans la dernière partie de ce chapitre, nous étudions une classe de détecteurs (notée Ω_x^z) qui généralise le détecteur Ω^z de la même façon qu'un détecteur $\diamond\mathcal{S}_x$ généralise la classe $\diamond\mathcal{S}$. Plus précisément, un détecteur Ω_x^z fournit à chaque processus un ensemble de leaders. Comme dans la spécification de $\diamond\mathcal{S}_x$, la portée de ce détecteur est limitée : seuls les processus appartenant à un ensemble Q de cardinal au moins x perçoivent au bout d'un certain temps le même ensemble de leaders (paragraphe 4.5.2). Ainsi, lorsque $x = n$, nous retrouvons la spécification des détecteurs Ω^z. Nous nous intéresserons à la calculabilité du k-accord dans les modèles asynchrones équipés d'un détecteur Ω_x^z. Cette étude nous conduira à définir le modèle associé $IRIS(PR_{\Omega_x^z})$ ainsi qu'à construire une extraction de Ω_x^z dans ce modèle (algorithme 4.19). Il s'avère que les propriétés $PR_{\Omega_n^z}$ et PR_{Ω^z} sont identiques. Donc, s'il on choisit n comme valeur du paramètre x, nous obtenons une construction d'un détecteur Ω^z dans le modèle $IRIS(PR_{\Omega^z})$. Pour éviter de dupliquer le code, nous renvoyons à l'extraction générale présentée dans le paragraphe 4.5.2.2. La proposition suivante se déduit de la correction de cet algorithme :

Proposition 4.6 *Il existe un algorithme qui construit un détecteur de défaillances de la classe Ω^z dans le modèle $IRIS(PR_{\Omega^z})$.*

4.4 Caractérisation

Nous avons montré comment simuler le modèle $IRIS(PR_{\mathcal{C}})$ dans le modèle $\mathcal{SM}_{n,n-1}[\mathcal{C}]$. Étant donné un détecteur de défaillances $\mathcal{C} \in \{\Omega^z, \diamond\mathcal{S}_x, \diamond\psi^y\}$, nous avons vu qu'il est possible d'exprimer les propriétés de ce détecteur comme une restriction des exécutions du modèle IIS. Réciproquement, nous avons montrer qu'à partir de la restriction $IRIS(PR_{\mathcal{C}})$, il est possible d'extraire un détecteur de la classe $\mathcal{C}$.

Quelle sont les puissances de calcul relatives des deux modèles ? On souhaiterait montrer l'équivalence des deux modèles, c'est à dire un théorème de la forme : Étant donné un problème de décision T, il existe une solution dans le modèle $IRIS(PR_{\mathcal{C}})$ si et seulement si il existe une solution dans $\mathcal{SM}_{n,n-1}[\mathcal{C}]$. Pour répondre à cette question, nous cherchons des simulations générales qui permettent de passer d'un modèle à l'autre, i.e.,

1. Dans le modèle $IRIS(PR_{\mathcal{C}})$, nous savons simuler l'opération « lire la sortie du détecteur » du modèle $\mathcal{SM}_{n,n-1}[\mathcal{C}]$. Comment simuler les opérations lire/écrire ?
2. Inversement, nous avons vu qu'il est possible de simuler l'opération restricted_w_snap() du modèle $IRIS(PR_{\mathcal{C}})$ dans le modèle $\mathcal{SM}_{n,n-1}[\mathcal{C}]$. Cependant, dans une telle simulation, les notions de processus corrects dans les deux modèles ne coïncident pas nécessairement. Un processus correct du point de vue du modèle $\mathcal{SM}_{n,n-1}[\mathcal{C}]$ peut être considéré défaillant par omission dans l'exécution $IRIS(PR_{\mathcal{C}})$ simulée.

Cette partie traite des deux problèmes ci-dessus. La section 4.4.1 décrit une simulation des opérations lire/écrire dans le modèle $IRIS(PR_{\mathcal{C}})$. L'algorithme étend la simulation de Borowsky et Gafni [24, 23] (qui ne traite que de l'équivalence entre les modèles *IIS* et $\mathcal{SM}_{n,n-1}[\emptyset]$). Dans la section 4.4.2, nous établissons que pour une classe restreinte de problèmes, les deux modèles ont la même puissance de calcul.

4.4.1 Simulation de $\mathcal{SM}_{n,n-1}[\mathcal{C}]$ dans $IRIS(PR_{\mathcal{C}})$

Dans $\mathcal{SM}_{n,n-1}[\mathcal{C}]$, l'exécution d'un algorithme consiste, pour chaque processus, en une suite de calculs locaux et d'opérations de base du modèle :

- Écrire une valeur v en mémoire partagée : write(v) ;
- Obtenir une vue de la mémoire partagée : snap() ;
- Lire la sortie du détecteur sous-jacent : fd_query().

Sans perte de généralités, nous supposons que la $k$ième valeur écrite par p_i est l'entier k [24]. Ainsi, la vue de la mémoire partagée retournée par une opération snap() sera un vecteur de n entiers.

Le problème consiste à concevoir dans le modèle $IRIS(PR_{\mathcal{C}})$ une opération simulate() qui prend en paramètre une opération à simuler $op \in \{\text{snap}(), \text{write}(), \text{fd\_query}()\}$ et retourne une valeur qui dépend du type de op. Dans une exécution admissible, les processus invoquent séquentiellement simulate(op). Il n'est pas permis d'effectuer un nouvel appel à simulate(op) tant que l'appel précédent n'est pas terminé.

La simulation est correcte si, dans toute exécution admissible, la spécification suivante est satisfaite. Pour le processus p_i, nous notons s_i un vecteur retourné par un appel simulate(snap()) et $(fd_i^1, fd_i^2, \ldots)$ la suite des valeurs renvoyées par les appels simulate(fd_query()).

Définition 4.2 (simulation de $\mathcal{SM}_{n,n-1}[\mathcal{C}]$ dans $IRIS(PR_{\mathcal{C}})$)

Comparaison : $\forall s_i, s_j : s_i \leq s_j \vee s_j \leq s_i$

Auto inclusion : $\forall s_i : s_i[i] = k$ *où* k *est le nombre d'appels* simulate(write(v)) *de* p_i *qui précèdent l'appel* simulate(snap()) *qui retourne* s_i

Progrès : $\forall s, \forall j : s[j] \geq k$ *où* k *est le nombre d'appels* simulate(write(v)) *de* p_j *qui précèdent l'appel* simulate(snap()) *qui retourne* s.

Terminaison : $\forall p_i$ *processus correct, tout appel* simulate(op) *effectué par* p_i *termine*

Cohérence : *L'ensemble des séquences* $(fd_i^1, fd_i^2, \ldots)_{i \in \Pi}$ *satisfont conjointement les propriétés de la classe* $\mathcal{C}$.

Notations

- Pour un ensemble de vecteurs $v_i \in \mathbb{N}^n$, $V = \{v_1, \ldots, v_\ell\}$, $\max_{cw}(V)$ désigne le maximum entrée par entrée des vecteurs de V. Formellement, $M = \max_{cw}(V)$ si et seulement si $\forall i \in \{1, \ldots, n\} : M[i] = \max\{v_1[i], \ldots, v_\ell[i]\}$.
- De même, $\min_{cw}(V)$ désigne le minimum entrée par entrée des vecteurs de V. Formellement, $m = \min_{cw}(V)$ si et seulement si $\forall i \in \{1, \ldots, n\} : m[i] = \max\{v_1[i], \ldots, v_\ell[i]\}$.
- Nous définissons une relation d'ordre partiel $\leq$ sur l'ensemble des vecteurs de $\mathbb{N}^n$ comme suit $v_1 \leq v_2$ si et seulement si $\forall i \in \{1, \ldots, n\} : v_1[i] \leq v_2[i]$.

Algorithme La Figure 4.15 décrit une implémentation de simulate(op). La technique employée est due à Borowsky et Gafni [23, 24]. L'algorithme se déroule par *phases*. Une phase ρ comprend deux rondes 2ρ et $2\rho - 1$ du modèle $IRIS(PR_{\mathcal{C}})$. L'algorithme se divise fonctionellement en deux parties :

- Simuler write(v)/snap().

  Durant la phase ρ, le processus p_i accède successivement aux objets $IS[2\rho - 1]$ et $IS[2\rho]$. p_i maintient à jour deux vecteurs de n entiers T_i (« *try* ») et C_i (« *commit* »). C_i représente un état *validé* de la mémoire partagée simulée, c'est à dire que C_i peut sereinement être retourné par p_i comme résultat de simulate(snap()). Autrement dit, lorsque $C_i[j] = k$ p_i sait que la $k$ième écriture de p_j est connue ou sera connue de tous les processus. Différemment, T_i représentent les écritures en cours mais qui ne sont pas encore nécessairement validées. Lorsque p_i débute la simulation de sa $k$ième écriture, il incrémente l'entrée i du vecteur $T_i[i]$ (ligne 1) pour annoncer cette écriture. Si, à la fin d'une phase, $C_i[j] < T_i[j] = k$ alors il existe des processus qui n'ont pas connaissance de la $k$ième écriture de p_j.

  Le comportement des processus au cours d'une phase ρ peut se décomposer comme suit. Le but de la première ronde est d'améliorer la connaissance des processus sur les écritures en cours. Chaque processus p_i poste son vecteur T_i (ligne 3) et calcule le maximum M_i entrée par entrée des vecteurs T qui apparaissent dans le snapshot immédiat obtenu (ligne 4). Lors de la deuxième ronde, les processus cherchent à identifier « la connaissance commune », c'est à dire les écritures qui sont ou seront connues de tous. p_i poste (ligne 6) ce qu'il appris lors de la première ronde (le vecteur M_i représente cette connaissance). Par rapport au snapshot immédiat sm_i obtenu, le maximum entrées par entrées des vecteurs $M \in sm_i$ est la connaissance maximale observable par p_i. En conséquence, T_i est mis à jour avec ce maximum (ligne 7). Par contre, le minimum entrée par entrée des $M \in sm_i$ est ou sera connu de tous. p_i peut donc considérer ce minimum comme un snapshot valide de la mémoire partagée simulée (ligne 8). En effet, si dans le vecteur minimum,

```
init : C_i[1:n] ← [0,...,0] ; T_i[1:n] ← [0,...,0] ; r_i ← 0 ;
       fd_ds_i ← fd_init()

operation simulate(op)
 (1)  if op = write() then T_i[i] ← T_i[i] + 1 endif ;
 (2)  repeat
 (3)      sm_i ← IS[r_i].restricted_w_snap(T_i, fd_ds_i) ; r_i ← r_i + 1 ;
 (4)      M_i  ← max_cw{T : T ∈ sm_i} ;
 (5)      < FD-OUTPUT_i, fd_ds_i > ← fd_update({fd_ds : fd_ds_i ∈ sm_i}) ;
 (6)      sm_i ← IS[r_i].restricted_w_snap(M_i, fd_ds_i) ;  r_i ← r_i + 1 ;
 (7)      T_i  ← max_cw{M : M ∈ sm_i} ;
 (8)      C_i  ← min_cw{M : M ∈ sm_i} ;
 (9)      < FD-OUTPUT_i, fd_ds_i > ← fd_update({fd_ds : fd_ds_i ∈ sm_i})
 (10) until (C_i[i] = T_i[i]) endrepeat ;
 (11) case op = fd_query() then return(FD-OUTPUT_i)
 (12)      op = snap()     then return(C_i)
 (13)      op = write()    then return()
 (14) endcase
```

FIG. 4.15 – Simulation de $\mathcal{SM}_{n,n-1}[\mathcal{C}]$ dans $IRIS(PR_{\mathcal{C}})$

l'entrée ℓ est égale à k alors, du fait de l'inclusion entre les snapshots immédiats sm_j, tout vecteur T_j est tel que $T_j[\ell] = k$ à l'issue de la phase ρ. Au plus tard lors de la phase suivante, tout vecteur M_j sera donc tel que $M_j[\ell] \geq k$ et par suite, $C_j[\ell] \geq k$. Enfin, p_i termine la simulation en cours lorsque sa dernière écriture (représentée par la valeur de l'entrée $T_i[i]$) est selon lui validée ($C_i[i] = T_i[i]$, ligne 10). Nous verrons dans la démonstration que lors de chaque phase au moins l'une des écritures en cours est validée.

– Extraire un détecteur de la classe $\mathcal{C}$.
En parallèle à leur utilisation pour simuler les opérations write(v) ou snap(), les objet $IS[r]$ servent de support à un algorithme $\mathcal{A}$ qui extrait un détecteur de la classe $\mathcal{C}$. Dans le modèle $IRIS(PR_{\mathcal{C}})$, la forme générique d'un tel algorithme est la suivante (voir par exemple les Figures 4.13 ou 4.14) :

```
fd_ds_i ← fd_init() ; r ← 1
repeat forever
    sm_i ← IS[r].restricted_w_snap(fd_ds_i) ; r ← r + 1 ;
    < FD-OUTPUT_i, fd_ds_i > ← fd_update({fd_ds : fd_ds_i ∈ sm_i})
endrepeat
```

La structure de données fd_ds_i ainsi que les fonctions fd_update() et fd_init() dépendent de la classe du détecteur extrait. A chaque ronde, p_i poste fd_ds_i et calcule (1) la nouvelle valeur de cette variable et (2) la nouvelle valeur de la sortie du détecteur en fonction de ce qu'il observe dans son snapshot immédiat sm_i. Dans la simulation générale, chaque appel $IS[r]$.restricted_w_snap() est utilisé pour mettre à jour fd_s_i et FD_OUTPUT$_i$ conformément à l'algorithme d'extraction $\mathcal{A}$. Le résultat d'une opération simulate(fd_query)() est simplement la dernière

valeur de FD_OUTPUT$_i$.

Démonstration Rappelons qu'une phase ρ s'étale sur deux rondes $2\rho-1$ et 2ρ et comprend les appels succesifs aux objets $IS[2\rho-1]$ et $IS[2\rho]$. V_i^ρ désigne la valeur du vecteur V sur le processus p_i après modification éventuelle lors de la phase ρ. Étant donnée une phase ρ, $smin^1$ (respectivement $smin^2$) désigne le plus snapshot immédiat sm renvoyé par les appels $IS[2\rho-1]$.restricted_w_snap() (respectivement $IS[2\rho]$.restricted_w_snap()).

Le lemme qui suit rassemble les propriétés élémentaires de l'algorithme concernant les relations d'ordre entre les vecteurs C_i et T_j.

Lemme 4.3 *$\forall i,j$, $\forall\rho$ phase :*

1. $\forall\rho,\forall i,j : (C_i^\rho \leq C_j^\rho) \vee (C_j^\rho \leq C_i^\rho)$
2. $\forall\rho,\forall i,j : C_i^\rho \leq T_j^\rho$
3. $\forall\rho,\forall i,j : C_i^\rho \leq C_j^{\rho+1}$

Démonstration Les preuves de ces propriétés consistent à examiner attentivement le déroulement du code.

Démonstration de la propriété 1 A l'issue de la deuxième ronde la phase ρ, $(sm_i \subseteq sm_j) \vee (sm_j \subseteq sm_i)$. Supposons sans perte de généralités que $sm_i \subseteq sm_j$. Il vient $C_j^\rho = \min_{cw}\{M : M \in sm_j\} \leq \min_{cw}\{M : M \in sm_i\} = C_i^\rho$.

Démonstration de la propriété 2 D'après le texte du protocole (lignes 8-10) et le fait que les vues sm_i sont ordonnées par inclusion, il vient :

1. $C_i^\rho \leq \min_{cw}\{M : M \in smin^2\}$;
2. $\max_{cw}\{M : M \in smin^2\} \leq T_j^\rho$.

Nous en déduisons que $C_i^\rho \leq T_j^\rho$.

Démonstration de la propriété 3 A l'issue de la phase ρ, les vecteurs C_i^ρ sont ordonnées (propriété 1). Soit $C_{\max}^\rho$ le plus grand d'entre eux. Nous avons

- $C_i^\rho \leq C_{\max}^\rho$;
- $\forall k : C_{\max}^\rho \leq T_k^\rho$ (propriété 2) ;
- Au début de la phase $\rho+1$, l'incrémentation éventuelle de $T_k[k]$ (ligne 1) ne change pas l'inégalité précédente ;
- $\forall k : \max_{cw}\{T : T \in sm_k\} = M_k^{\rho+1}$ (ligne 4) ;
- $\min_{cw}\{M^{\rho+1} : M^{\rho+1} \in sm_j\} = C_j^{\rho+1}$.

En suivant dans l'ordre les (in)égalités ci dessus, il vient $C_i^\rho \leq C_j^{\rho+1}$. $\square_{Lemme\ 4.3}$

Le lemme suivant établit que la simulation produit, du point de vue du résultat des opérations simulées write()/snap(), une exécution valide dans le modèle mémoire partagée. op_x désigne une opération write() ou snap(). La valeur du vecteur C au moment de la terminaison de la simulation de op_x est noté s_x. Si op_x = snap(), s_x est donc le snapshot de la mémoire partagée simulée retourné en résultat de simulate(snap()).

Lemme 4.4 *Pour une opération $op_x, x \in \{1,2\}$ simulée par le processus $p_{(op_x)}$, soit s_x la valeur du vecteur C sur ce processus lorsque la simulation termine. En supposant que la simulation de op_x démarre à la phase ρ_x^s et termine à la phase ρ_x^e, nous avons :*

1. $(s_1 \leq s_2) \vee (s_2 \leq s_1)$;
2. $\rho_e^1 \leq \rho_s^2 \Rightarrow s_1 \leq s_2$;
3. *Si op_x est la kième écriture simulée par p_i alors $s_x[i] = k$;*
4. *Si la simulation de la kième écriture de p_j n'a pas commencé avant la phase ρ_e^x alors $s_x[j] < k$.*

Démonstration

- Les propriétés 1 et 2 découlent immédiatement des points 1 et 3 du Lemme 4.3 ;
- Propriété 3. Remarquons qu'à chaque nouvelle invocation de simulate(write(v)), p_i incrémente $T_i[i]$ (ligne 1) et qu'aucun processus $\neq p_i$ n'incrémente l'entrée i de son vecteur T (Autrement dit, $T_j^\rho[i] \geq \alpha$ implique que p_i a démarré la simulation de sa αième écriture lors d'une phase $\leq \rho$). D'où $s_x[i] = C_i[i] = T_i[i] = k$ lorsque l'invocation simulate(op_x) (ligne 10).
- Propriété 4. Pour le processus p_i, $C_i^\rho[j] \geq k$ implique que p_j a démarré la simulation de sa $k$ième écriture lors d'une phase $\leq \rho$. Or la simulation de l'écriture de rang k (si elle existe) démarre à une phase $> \rho_e^x$. D'où $s_x[j] = C_i[x] < k$ lorsque p_i termine la simulation de op_x.

$\square_{Lemme\ 4.4}$

Cette partie de la démonstration s'intéresse à la vivacité de la simulation. Nous montrons que chacune des invocations simulate(op) effectués par les processus corrects termine (Lemme 4.5) tandis que les processus défaillants ne peuvent simuler qu'un nombre fini d'opérations write(v). En particulier, un processus défaillant p_i par omission ne peut changer l'état de la mémoire simulée qu'un nombre fini de fois (Lemme 4.6). Ce dernier lemme montre que l'état de la mémoire simulée et la sortie du détecteur extrait sont bien cohérents. Du point de vue du détecteur extrait, le processus p_i est défaillant : à partir d'un certain temps, il ne doit plus agir de façon visible sur la mémoire partagée simulée.

Lemme 4.5 *Soit simulate(op) une opération initiée par p_i. Si p_i est correct alors l'opération termine.*

Démonstration Nous supposons d'abord que $op \in \{\text{snap}(), \text{fd\_query}()\}$. Soit α le rang de la dernière écriture terminée par p_i ($\alpha = 0$ si p_i n'a pas précédemment simulé d'écritures). A la fin de la simulation de cette écriture, $C_i[i] = T_i[i] = \alpha$. Observons également que tant que p_i n'a pas initié la simulation de l'écriture de rang $(\alpha + 1)$, $\forall j : M_j[i] \leq \alpha, C_i[j] \leq \alpha$ et $T_j[i] \leq \alpha$ (ligne 1). Par ailleurs, les suites $(C_i^p[i])$ et $(T_i^\rho[i])$ sont croissantes.

A l'issue de la phase à laquelle démarre la simulation de op, nous avons donc d'après les observations qui précèdent, $C_i[i] = T_i[i] = \alpha$. Le prédicat de terminaison de la boucle **repeat** est satisfait et l'appel simulate(op) termine.

Considérons maintenant l'écriture de rang α de p_i et supposons que cette écriture ne termine pas. On sait qu'il existe une ronde à partir de laquelle p_i observe dans sm_i uniquement des données postées par des processus corrects (propriété 4.1). Donc, à partir d'une certaine phase ρ_1, $T_j \in sm_i$ (ligne 4) ou $M_j \in sm_i$ (ligne 6) implique $p_j \in Correct$.

Nous montrons qu'à partir d'une certaine phase ρ_2, tous les vecteurs T_j postés (ligne 4) par les processus corrects sont tels que $T_j[i] \geq \alpha$. Remarquons d'abord que, puisque l'αième écriture de p_i ne termine pas, à partir d'une certaine phase $T_i[i] \geq \alpha$ et $\forall j : T_j[i] \leq \alpha$ (ligne 1 et seul p_i peut introduire une nouvelle valeur dans l'entrée i des vecteurs T).

Soient p_x et p_y deux processus et ρ une phase telle que (1) $T_x[i] = \alpha$ au début de cette phase et (2) p_y observe les données (T_x ou M_x) au cours de ρ. Alors, à l'issue de ρ, $T_y[i] = \alpha$. Nous notons P cette propriété.

- Si p_y observe p_x lors de la première ronde de la phase ρ ($T_x \in sm_y$) alors $M_y^\rho[i] \geq \alpha$ (ligne 4). Par suite, $T_y^\rho[i] = \alpha$ car, lors de la deuxième ronde, $M_y \in sm_y$ (propriété d'auto-inclusion) , $T_y^\rho[i]$ est le maximum des $M[i]$ contenu dans sm_y (ligne 7) et les $T[i]$ sont bornés par α.
- Si p_y observe p_x lors de la deuxième ronde de la phase ρ ($M_x \in sm_y$) alors $\alpha \geq T_y^\rho[i] \geq M_x^\rho[i]$ (ligne 7). Par suite, $T_y^\rho[i] \geq \alpha$ car, d'après le calcul de M_x lors de la première ronde, $M_x^\rho[i] \geq T_x[i] \geq \alpha$ (ligne 4 et propriété d'auto-inclusion des vues sm).

Soit maintenant un processus correct p_j. Par définition, $p_j \overset{s}{\rightsquigarrow} p_i$, c'est à dire que p_j observe infiniment souvent directement ou indirectement p_i. Par conséquent il suit de la propriété P qu'à partir d'une certaine phase, $T_j[i] = \alpha$. Ceci est vrai pour tout processus correct. Il existe donc une phase ρ_2 à partir de laquelle tous les processus corrects p_j sont tels que $T_j[i] = \alpha$.

Soit $\rho > \max(\rho_1, \rho_2)$. Déroulons le code pour calculer la valeur de $C_i[i]$ à l'issue de cette phase. Les vecteurs T ou M observés par p_i sont écrits par des processus corrects ($\rho > \rho_1$). Donc $C_i[i] = \min\{M^\rho[i] : M^\rho \in S\}$ où S est un sous ensemble des processus corrects. Or pour un processus correct p_j, $M_j^\rho[i] = \alpha$. En effet, lors de la première ronde sm_j contient $T_j^{\rho-1}[i] = \alpha$ et les $T[i]$ sont bornés par α. Par conséquent $C_i^\rho[i] = \alpha$ et le prédicat de sortie de la boucle **repeat** est satisfait. La αième écriture de p_i termine : contradiction. $\square_{Lemme\ 4.5}$

Lemme 4.6 *Soit p_i un processus fautif. Le nombre d'opérations* simulate(write(v)) *complétées par p_i est fini.*

Démonstration Si p_i est défaillant par panne franche, il effectue un nombre fini d'appels simulate(write(v)) et le lemme est trivialement vrai. Supposons que p_i soit défaillant par omission et que le nombre d'opérations simulate(write(v)) menées à leur terme par p_i soit non borné.

D'après la notion de processus correct dans $IRIS(PR_{\mathcal{C}})$, il existe une phase ρ_0 à partir de laquelle les processus corrects n'observent pas p_i. Plus précisément, $\forall \rho > \rho_0, \forall p_j \in Correct$, les ensembles sm_j calculés lors de la phase ρ ne contiennent pas T_i

ou M_i. Supposons que la αième écriture de p_i démarre à la phase $\rho_1 > \rho_0$ (c'est à dire que p_i exécute $T_i[i] \leftarrow \alpha$ à la ligne 1 au début de ρ).

Nous montrons que les processus corrects n'apprennent jamais que p_i essaie d'effectuer sa αième écriture. Formellement, $\forall \rho \geq \rho_1 : (\forall p_j \in \mathit{Correct} : T_j^{\rho-1}[i] < \alpha) \Rightarrow (\forall p_j \in \mathit{Correct} : T_j^{\rho}[i] < \alpha$ et $M_j^{\rho}[i] < \alpha)$. Soit p_j un processus correct et ρ une phase qui satisfait le prémisse de l'implication.

- $M_j^{\rho}[i] = \max\{T_x^{\rho-1}[i] : p_x \in S\}$ où S est sous-ensemble de $\mathit{Correct}$. Ceci vient du code (ligne 4) et du fait que p_j n'observe que des processus corrects lors de la phase ρ. Par conséquent, $\forall p_j \in \mathit{Correct} : M_j^{\rho}[i] < \alpha$.
- De même, $T_j^{\rho}[i] = \max\{M_x^{\rho}[i] : p_x \in S\}$ où S est sous-ensemble de $\mathit{Correct}$. D'où $\forall p_j \in \mathit{Correct} : T_j^{\rho}[i] < \alpha$.

Calculons maintenant la valeur de $C_i^{\rho}[i]$ pour $\rho \geq \rho_1$. Nous savons que $C_i^{\rho}[i] \leq \min\{M^{\rho}[i] : M^{\rho} \in \mathit{smin}_2\}$ (ligne 8). Or après ρ_0, la plus petite vue smin_2 ne peut contenir que des vecteurs M écrits par des processus corrects. Comme $\rho_0 < \rho_1 \leq \rho$, nous en déduisons à l'aide de la propriété ci-dessus que $C_i^{\rho}[i] < \alpha = T_i^{\rho}[i]$. Ceci est vrai pour toute phase $\geq \rho_1$. Par conséquent la αième écriture de p_i ne termine pas.

$\square_{\mathit{Lemme}\ 4.6}$

Proposition 4.7 *Soient $\mathcal{C}$ une classe de détecteurs de défaillances et $IRIS(PR_{\mathcal{C}})$ une restriction du modèle IIS. Supposons qu'il existe un algorithme $\mathcal{E}$ qui construise un détecteur de la classe $\mathcal{C}$ dans $IRIS(PR_{\mathcal{C}})$. Soit T un problème de décision. S'il existe une solution à T dans le modèle $\mathcal{SM}_{n,n-1}[\mathcal{C}]$ alors il existe une solution à T dans le modèle $IRIS(PR_{\mathcal{C}})$.*

Démonstration Soit $\mathcal{A}$ un algorithme qui résout T dans le modèle $\mathcal{SM}_{n,n-1}[\mathcal{C}]$. Chaque processus p_i effectue le code décrit par l'algorithme $\mathcal{A}$ en simulant les opérations write()/snap() ou consultation du détecteur à l'aide de simulate(). Lorsque p_i atteint le terme de $\mathcal{A}$, il rentre dans une boucle sans fin au cours de laquelle il invoque infiniment souvent simulate(fd_query()).

Nous devons montrer que les vues successives de la mémoire partagée simulée prennent en compte les écritures effectuées et sont cohérentes avec les valeurs successives fournies par le détecteur. Il faut par exemple s'assurer qu'un processus défaillant ne n'écrive pas après l'instant auquel il défaille. Dans le cas contraire, la sortie du détecteur et l'état de la mémoire simulé ne serait pas cohérent. Pour montrer cela, nous avons besoin de dater les évènements de l'exécution simulée.

Les évènements pertinents sont le début et la fin des opérations write(v), snap() et fd_query(). Pour une opération op, nous notons τ_{op}^s et τ_{op}^e les dates de début et de fin de l'opération op. Un appel simulate(op) se déroule sur un certain nombre de phase. τ_{op}^s et τ_{op}^e sont définies comme les numéros de phase auxquelles la simulation de op démarre et s'arrête respectivement ($\tau_{op}^e = +\infty$ si l'invocation de simulate(op) ne termine jamais). Muni de cette notion de date, nous pouvons maintenant définir dans l'exécution simulée l'ensemble des processus correct $\mathit{Correct}_{rw}$ et les instants des défaillances des processus fautifs.

- $Correct_{rw}$ est l'ensemble des processus corrects (du point de vue de l'exécution dans le modèle $IRIS(PR_{\mathcal{C}})$).
- $Fautif_{rw}$ est donc l'ensemble des processus qui défaillent par panne franche ou par omission du point de vue de l'exécution dans le modèle $IRIS(PR_{\mathcal{C}})$. Quelles sont les dates des défaillances dans l'exécution simulée ?
  Soit ρ_c la plus petite phase à partir de laquelle les processus corrects n'observent que des processus corrects. Plus précisément, $\forall r \geq 2\rho_c - 1, \forall p_i \in Correct : j \in sm_i^r \Rightarrow p_j \in Correct$. La propriété 4.1 assure qu'une telle phase existe. Remarquons que pour $p_i \in Correct$, l'exécution $IRIS(PR_{\mathcal{C}})$ courante est indistinguable d'une exécution dans laquelle tous les processus $p_j \notin Correct$ défaillent par panne franche juste avant le début de la phase ρ_c. Ainsi, l'instant τ_c des défaillances dans l'exécution est la date ρ_c.

Le Lemme 4.4 implique qu'il existe une linéarisation des opérations simulées write(v), snap() et fd_query() qui respecte leurs dates de début et de fin. Dans une telle linéarisation, chaque snapshot est égal à ou plus grand que le snapshot qui le précède et contient $\forall j$ la dernière écriture de p_j qui le précède. De plus, chaque processus $p_i \in Correct_{rw}$ possède une infinité d'opérations.

Pour terminer la démonstration, nous devons nous assurer que l'état de la mémoire partagée (observé à travers le résultat d'invocations simulate(snap())) et les valeurs successives du détecteur sont cohérentes avec le motif des défaillances. Il faut donc montrer que (*C1*) chaque processus $\in Fautif_{rw}$ n'écrit pas en mémoire partagée après la date τ_c à laquelle il défaille et (*C2*) les valeurs retournées par les appels simulate(fd_query()) satisfont la spécification de la classe $\mathcal{C}$ pour le motif de défaillance dans lequel tous les processus fautifs défaillent à la date τ_c.

Démonstration de C1 Soient $p_j \in Fautif_{rw}$ et $k = \max\{T_i^{\rho_c}[j] : p_i \in Correct_{rw}\}$. Puisqu'aucun processus correct n'observe p_j au cours de la phase ρ_c, l'écriture de rang k de p_j a démarré lors d'une phase $\leq \rho_c$, c'est à dire avant le crash de p_j.

$\forall \rho \geq \rho_c, \forall p_i \in Correct_{rw} : T_i^{\rho}[j] \leq k$ car les processus corrects observent uniquement des processus corrects à partir de la phase ρ_c. Soit s un le résultat d'une opération snap() qui termine à la date $\rho \geq \rho_c$. Lorsque l'opération termine, $s[j] = C_x^{\rho}[j]$ où p_x est le processus qui effectue l'opération. Notons $smin^2$ le plus petit snapshot immédiat retourné lors de la deuxième phase de la ronde ρ. D'après le code, $C_x^{\rho} \leq \min_{cw}\{M^{\rho} : M^{\rho} \in smin^2\}$. Or lors de la phase ρ, les processus corrects n'observent que des processus corrects. D'où $M^{\rho} \in smin^2 \Rightarrow \exists p_i \in Correct_{rw}$ tel que $M^{\rho} = M_i^{\rho}$. De plus notons que $M_i^{\rho} = \max_{cw}\{T^{\rho-1} : T^{\rho-1} \in S\}$ avec S un sous ensemble des processus correct. Il vient $s[j] \leq k$ car pour tout processus correct, nous avons toujours $T[j] \leq k$. Ainsi, p_j ne peut écrire après la date de sa défaillance. *Fin de la démonstration de C1*

Démonstration de C2 Pour les processus corrects, l'exécution $IRIS(PR_{\mathcal{C}})$ est indistinguable d'une exécution dans laquelle les tous les processus fautifs sont défaillants par panne franche au début de la phase ρ_c. Puisque l'extraction $\mathcal{E}$ est correcte, les valeurs successives retournées par les appels simulate(fd_query()) sont admissibles pour la spécification de $\mathcal{C}$ dans une exécution dans laquelle tous les processus fautifs défaillent à la

date ρ_c. *Fin de la démonstration de C2* $\square_{Proposition\ 4.7}$

4.4.2 Caractérisation

La proposition 4.7 montre que tout problème de décision calculable dans le modèle $\mathcal{SM}_{n,n-1}[\mathcal{C}]$ l'est aussi dans le modèle $IRIS(PR_{\mathcal{C}})$ à condition que l'on dispose d'une extraction d'un détecteur de la classe $\mathcal{C}$ dans le modèle $IRIS(PR_{\mathcal{C}})$.

Réciproquement, supposons que l'on dispose d'un algorithme $\mathcal{A}$ qui résout une tâche quelconque T dans le modèle $IRIS(PR_{\mathcal{C}})$. Nous avons vu que pour $\mathcal{C} \in \{\Omega^z, \diamond\mathcal{S}_x, \diamond\psi^y\}$, nous pouvons simuler les opérations restricted_w_snap() du modèle $IRIS(PR_{\mathcal{C}})$ dans le modèle $\mathcal{SM}_{n,n-1}[\mathcal{C}]$. Cependant, l'analyse du comportement des processus dans les exécutions du modèle $IRIS(PR_{\mathcal{C}})$ montre que certains processus sont défaillants (du point de vue du modèle $IRIS(PR_{\mathcal{C}})$) bien qu'ils effectuent une infinité d'opérations restricted_w_snap(). L'algorithme $\mathcal{A}$ ne mène pas nécessairement ces processus à décider. Néanmoins, les processus qui défaillent par omission observe le déroulement de l'exécution des processus corrects. Ils peuvent donc en déduire la décision de ces processus. Cette observation nous amène à considérer une classe restreinte de problèmes : les problème de *d'accord*.

Problème d'accord Un problème d'accord est une tâche de décision dans laquelle tout processus peut calculer localement sa décision à partir de la connaissance de la décision d'un autre processus. Le consensus, le (n,k)-accord ou le (n,k)-test&set sont par exemple des problèmes d'accord. Le théorème qui suit établit que pour tout problème d'accord, il existe une solution dans le modèle $\mathcal{SM}_{n,n-1}[\mathcal{C}]$ si et seulement il existe une solution dans le modèle associé $IRIS(PR_{\mathcal{C}})$.

Théorème 4.1 *Soit $\mathcal{C}$ une classe de détecteur de défaillance et $IRIS(PR_{\mathcal{C}})$ une restriction du modèle IIS tels qu'il existe une simulation $\mathcal{S}$ du modèle $IRIS(PR_{\mathcal{C}})$ dans $\mathcal{SM}_{n,n-1}[\mathcal{C}]$ et une extraction $\mathcal{E}$ d'un détecteur de la classe $\mathcal{C}$ dans le modèle $IRIS(PR_{\mathcal{C}})$. Pour tout problème d'accord T, il existe un algorithme pour T dans $\mathcal{SM}_{n,n-1}[\mathcal{C}]$ si et seulement si il existe un algorithme qui résout T dans le modèle $IRIS(PR_{\mathcal{C}})$.*

Démonstration

$\Rightarrow$ Soit $\mathcal{A}$ un algorithme qui résout T dans le modèle $IRIS(PR_{\mathcal{C}})$. Pour résoudre T dans le modèle $\mathcal{SM}_{n,n-1}[\mathcal{C}]$, les processus effectue l'algorithme décrite dans la figure ci dessous.

Le rôle de la tâche $T1$ est de calculer en simulant $\mathcal{A}$ via $\mathcal{S}$ une décision. Lorsque la simulation de $\mathcal{A}$ termine, p_i écrit la décision obtenue en mémoire partagée. Dans la tâche $T2$, p_i observe les décisions éventuelles obtenues par les autres processus. Si il existe j tel que $DEC[j] \neq \perp$, alors p_i est capable de calculer localement une décision valide car T est un problème d'accord. Les processus exécutent les tâches $T1$ et $T2$ en parallèle.

L'algorithme $\mathcal{A}$ résout T dans le modèle $IRIS(PR_{\mathcal{C}})$ et la simulation $\mathcal{S}$ est réputée correcte. Il existe donc au moins un processus correct qui décide en exécutant la

init $DEC[1:n] \leftarrow [\perp, \ldots, \perp]$
  $dec_i \leftarrow \perp$ % write once decision variable

Task $T1$:
  simulate $\mathcal{A}$ with the simulation $\mathcal{S}$;
  $dec_i \leftarrow$ final decision output by the simualtion of $\mathcal{A}$;
  $DEC[i] \leftarrow dec_i$

Task $T2$:
  repeat foreach $j \in \{1, \ldots, n\}$ **do** $view_i[j] \leftarrow DEC[j]$ **enddo**
  until $(\exists j \in \{1, \ldots, n\} : view_i[j] \neq \perp)$;
  if $dec_i = \perp$ **then** $dec_i \leftarrow$ compute locally a valid decision from $view_i[j] \neq \perp$ **endif**

FIG. 4.16 – Résoudre T dans le modèle $\mathcal{SM}_{n,n-1}[\mathcal{C}]$

tâche $T1$. Par conséquent, il existe inéluctablement une entrée $\neq \perp$ dans le tableau partagé DEC. Ainsi, un processus correct décide ou bien directement en simulant $\mathcal{A}$ ou alors en calculant localement une décision valide à partir de la décision d'un autre processus observée dans le tableau DEC.

$\Leftarrow$ Cette implication est la proposition 4.7.

$\square_{Théorème\ 4.1}$

4.5 Applications

Pour illustrer l'intérêt de l'approche développée dans les paragraphes précédents, nous présentons deux applications. La première consiste en un algorithme simple qui résout le (n,k)-accord dans le modèle $IRIS(PR_{\Omega^z})$ pour $k \geq z$. La seconde est une borne sur la calculabilité du (n,k)-accord dans le modèle à mémoire muni de détecteurs à portée limitée.

Il est connu que les classes $\diamond\mathcal{S}$ et Ω sont équivalentes [29]. La classe $\diamond\mathcal{S}_x$ généralise la classe $\diamond\mathcal{S}_x$ en affaiblissant la propriété de précision : seul un sous ensemble des processus doit faire confiance à un même processus correct. S'il l'on considère la version représentant de la spécification de cette classe, un détecteur $\diamond\mathcal{S}_x$ peut être vu comme un détecteur Ω à portée limitée : dans un sous ensemble Q du système, tous les processus possède le même représentant, c'est à dire le même leader. Ces observations nous amènent à définir la variante à portée limitée de la classe Ω^z que l'on notera Ω^z_x. De même qu'un détecteur Ω^z, un détecteur Ω^z_x fournit à chaque processus un ensemble d'identités. A partir d'un certain temps, chacun de ces ensembles contient toujours l'identité d'au moins un processus correct. Cependant, à la différence de Ω^z, la propriété de leader commun inéluctable ne s'étend pas à tout le système mais est restreinte à un sous ensemble du système, exactement comme dans la spécification de $\diamond\mathcal{S}_x$.

Dans [71], Herlihy et Penso généralisent la classe $\diamond\mathcal{S}_x$ en divisant les processus en q

ensembles disjoints $X_1, \ldots, X_q$. Chaque sous ensemble X_i est vu comme un sous système muni d'un détecteur $\diamond\mathcal{S}_x$. Adoptant cette stratégie, nous définissons de la même façon le détecteur « partionné » $\Omega^z_{x,q}$ (paragraphe 4.5.2).

Dans le modèle à mémoire partagée muni d'un détecteur $\Omega^z_{x,q}$, quel est le plus petit k tel qu'il existe un algorithme qui résout le problème du (n,k)-accord ? Nous montrons qu'il existe une solution $(n-1)$-résiliente si et seulement si $k \geq n - x + qz$ (théorème 4.2). Par contraste avec la démonstration d'un résultat similaire pour le détecteur $\diamond\mathcal{S}_x$ qui repose sur des outils issus de la topologie, la preuve proposée ici repose sur une analyse simple des exécutions du modèle associé $IRIS(PR_{\Omega^z_{x,q}})$.

4.5.1 (n,z)-accord dans $IRIS(PR_{\Omega^z})$

Dans ce paragraphe, nous présentons un algorithme pour résoudre le problème du (n,z)-accord dans le modèle $IRIS(PR_{\Omega^z})$. L'algorithme se divise en deux parties (z-adopt() et z-converge(), cf. Figure 4.17). Il combine dans le modèle $IRIS(PR_{\Omega^z})$ les techniques simples proposées dans [122] et [4]. La première partie assure que les processus corrects décident (vivacité) tandis que la seconde garantit la sûreté (c'est à dire au plus z valeurs distinctes sont décidées).

```
init r_i ← 1 ; est_i ← v_i ; dec_i ← ⊥ % write-once decision variable

loop forever
 (1)  est_i ← z-adopt(est_i, r_i) ;
 (2)  r_i ← r_i + (z + 1) ; % z-adopt() uses z + 1 consecutive IS objects %
 (3)  < est_i, commit_i > ← z-converge(est_i, r_i) ;
 (4)  r_i ← r_i + 2 ; % z-converge() uses 2 consecutive IS objects %
 (5)  if (commit_i) ∧ (dec_i = ⊥) then dec_i ← est_i end if
end loop
```

FIG. 4.17 – (n,z)-accord dans $IRIS(PR_{\Omega^z})$

L'algorithme procède par phases successives. A chaque phase est associée un bloc de $z+3$ objets $IS[r]$ consécutifs. Les $z+1$ premiers objets sont utilisés par z-adopt() et les deux autres par z-converge(). Le but de l'appel à z-adopt() est de réduire le nombre de valeurs présentes initialement au début de la ronde à au plus z. Ensuite, en appelant z-converge(), les processus vérifient s'il est sûre de décider, c'est à dire si le nombre de valeurs encore présente dans l'exécution est inférieur ou é gale à z.

L'algorithme z-adopt() Chaque invocation de l'opération z-adopt() prend en paramètre une valeur v et retourne une valeur v' telle qu'il existe une invocation z-adopt(v'). De plus, si les processus exécutent une suite infinie d'invocations, il existe un rang à partir duquel les appels z-adopt() retournent au plus z valeurs distinctes. Formellement, les valeurs de retournés par z-adopt() satisfont les propriétés suivantes :

– *Terminaison* : Pour tout processus correct, tout appel z-adopt() termine ;

- *Validité* : Si un processus obtient v comme résultat de z-adopt(), il existe un processus qui a invoqué z-adopt(v) ;
- z-*adoption* : Soit $(inv1, inv2, \ldots)$ une suite infinie d'appels à z-adopt(). Il existe un rang r dans cette séquence tel que tous les appels de rang $r' \geq r$ retournent au plus z valeurs distinctes.

Comment implémenter cette spécification dans le modèle $IRIS(PR_{\Omega^z})$? Nous savons qu'il existe un ensemble L , $|L| \leq z$ et une ronde R à partir de laquelle tous les plus petits snapshots retournés par les objets $IS[r]$ $(r \geq R)$ sont contenus dans L. D'autre part nous avons vue comment identifier de tels snapshots minimaux (algorithme get_smin(), paragraphe 4.1.2). Ainsi, les processus identifient un plus petit snapshot *smin* récent en exécutant une variante appropriée de l'algorithme get_smin() et choisissent ensuite une valeur arbitraire dans ce snapshot qui est retourné en résultat de z-adopt(). Lorsque la propriété PR_{Ω^z} est satisfaite, ces snapshots $smin^r$ sont inclus dans un ensemble de cardinal au plus z. Par conséquent, au plus z valeurs distincts sont alors retournées.

L'algorithme z-converge() L'opération z-converge(v) prend en paramètre une valeur et retourne un couple $< c, v >$ où c est un booléen et v une valeur. Dans la lignée de [122], nous dirons qu'un processus *ferre* v si son appel retourne $(< c, v >)$. Lorsque $c = true$, nous dirons que ce processus *pêche* v. Les valeurs retournées par les appels z-converge() satisfont la spécification suivante [122] :

- *Terminaison* : Tout appel z-converge(v) par un processus correct termine ;
- *Validité* : Si un processus ferre v alors il existe un processus qui a invoqué z-converge() avec v en paramètre ;
- z-*accord convergeant* : Si un processus pêche une valeur v alors au plus z valeurs distinctes sont ferrées.
- z-*convergence* : Si le cardinal de l'ensemble des valeurs v passées en paramètre des appels z-converge() est borné par z, tous les processus qui terminent leur appel pêchent une valeur.

L'algorithme décrit dans la Figure 4.18 traduit l'algorithme originel dans le modèle $IRIS(PR_{\mathcal{C}})$. Nous renvoyons le lecteur à [122] où la correction de l'algorithme est démontrée.

operation z-converge(v_i)
(1) $est_i \leftarrow v_i$; $ok_i \leftarrow false$;
(2) $s_i \leftarrow IS[r]$.restricted_w_snap($< i, est_i >$) ; $r_i \leftarrow r_i + 1$;
(3) **if** $\big|\{est_j :< j, est_j >\in s_i\}\big| \leq k$ **then** $ok_i \leftarrow true$ **end if** ;
(4) $t_i \leftarrow IS[r_i]$.restricted_w_snap($< i, est_i, ok_i >$) ;
(5) **case** $\forall j :< j, est_j, ok_j >\in t_i : ok_j$ **then** $return(< est_i, true >)$
(6) $\exists j :< j, est_j, ok_j >\in t_i : ok_j$ **then** $return(< est_j, false >)$
(7) $\forall j :< j, est_j, ok_j >\in t_i : \neg ok_j$ **then** $return(< est_i, false >)$
(8) **end case**

FIG. 4.18 – z-converge algorithm (code for p_i)

4.5.2 Ω^z par groupes : la classe $\Omega^z_{x,q}$

Comme annoncé en introduction de cette partie, nous étudions la puissance de calcul additionnelle que l'on obtient en équipant le modèle de base avec un détecteur de la classe $\Omega^z_{x,q}$. Nous nous intéressons plus particulièrement au (n,k)-accord. On souhaite caractériser en fonction des paramètres n, q, z et x quelle est la plus petite valeur de k pour laquelle il existe une solution au problème (n,k)-accord. Dans le cas particulier $z = 1$, cette caractérisation est obtenue dans [71] pour le modèle $\mathcal{MP}_{n,t}[\Omega^1_{x,q}]$ à l'aide d'outils mathématiques puissants issus de la topologie algébrique. Par contraste, l'étude développée ici repose uniquement sur des réductions algorithmiques.

4.5.2.1 La classe $\Omega^z_{x,q}$ et la restriction $PR_{\Omega^z_{x,q}}$

La famille $\{\Omega^z_{x,q}\}_{1\leq x\leq n, 1\leq q\leq n, 1\leq z\leq n}$ La classe $\Omega^z_{x,q}$ étend la notion de détecteur à portée limitée dans un système où les processus sont regroupés en ensembles disjoints. Il existe q groupes disjoints notés $X_1, \ldots, X_q$ tels que $|\bigcup_{1\leq i\leq q} X_i| \geq q$. On note $x_i = |X_i|$ et $X = \bigcup_{1\leq i\leq q} X_i$ l'union des X_i. Dans chaque groupe X_i, il existe un ensemble de leaders L_i. A partir d'un certain temps, tous les processus de X_i font confiance au même ensemble de leaders $L_i \subseteq X_i$. Autrement dit, il existe un détecteur de la classe Ω^z dans chaque groupe X_i.

Plus précisément, un détecteur de la classe $\Omega^z_{x,q}$ contrôle sur chaque processus p_i une variable LEADER_i qui contient un ensemble d'identités de processus. A un instant donné, l'ensemble $\{p_j : j \in \text{LEADER}_i\}$ constitue l'ensemble des leaders de p_i. Un détecteur est dans la classe $\Omega^z_{x,q}$ si il satisfait les contraintes suivantes :

$$\begin{array}{rcl}
\exists X_1, \ldots, X_q & : & (\forall \alpha, \beta : X_\alpha \cap X_\beta = \emptyset) \wedge (x \leq |X_1 \cup \ldots \cup X_q|), \\
\exists L_1, \ldots, L_q & : & \forall \alpha : (1 \leq |L_\alpha| \leq z) \wedge (L_\alpha \subseteq X_\alpha), \\
\exists \tau \text{ tels que} & : & \forall \tau' : \tau \leq \tau', \forall i \in \Pi, \forall \alpha : 1 \leq \alpha \leq q \\
 & & (1)\ (X_\alpha \cap \mathit{Correct} \neq \emptyset) \Rightarrow (L_\alpha \cap \mathit{Correct} \neq \emptyset) \\
 & & (2)\ i \in X_\alpha \Rightarrow \text{LEADER}_i^{\tau'} = L_\alpha \\
 & & (3)\ i \notin X_\alpha \Rightarrow \text{LEADER}_i^{\tau'} = \{i\}
\end{array}$$

Lorsque $q = 1$ (un seul groupe), $x = n$ (le détecteur porte sur tout le système), nous retrouvons la définition de la classe Ω^z. De même, lorsque $q = 1$ et $z = 1$, la classe $\Omega^1_{x,1}$ se réduit à la classe $\diamond\mathcal{S}_x$. En effet, la spécification ci-dessus devient :

$$\begin{array}{rcl}
\exists X & : & x \leq |X|,\ \exists \ell \in \Pi : \ell \in X, \\
\exists \tau \text{ tels que} & : & \forall \tau' : \tau \leq \tau', \forall i \in \Pi \\
 & & (1)\ (X \cap \mathit{Correct} \neq \emptyset) \Rightarrow (\ell \in \mathit{Correct}) \\
 & & (2)\ i \in X \Rightarrow \text{LEADER}_i^{\tau'} = \ell \\
 & & (3)\ i \notin X_\alpha \Rightarrow \text{LEADER}_i^{\tau'} = \{i\}
\end{array}$$

ce qui correspond bien à la définition version représentant de la classe $\diamond\mathcal{S}_x$.

La restriction $PR_{\Omega^z_{x,q}}$ La propriété $PR_{\Omega^z_{x,q}}$ est semblable à la propriété PR_{Ω^z}. PR_{Ω^z} requiert qu'à partir d'une certain ronde R, le plus petit snapshot retourné est toujours contenu dans le même ensemble L de cardinal au plus z. Nous souhaitons définir de façon similaire la restriction du modèle *IIS* associée à la classe $\Omega^z_{x,q}$. Pour cela, étant donné un groupe X_α, nous considérons pour chaque ronde r le plus petit snapshot $smin^r_{|X_\alpha}$ obtenu par les processus de ce groupe. « Le plus petit snapshot est toujours inclus dans un même ensemble » devient « les identités de X qui apparaissent dans $smin^r_{|X_\alpha}$ appartiennent toujours au même ensemble L_α. Plus précisément, $\exists L_\alpha : |L_\alpha| \leq z, \exists R$ tels que $\forall r \geq R : smin^r(X_\alpha) \cap X_\alpha \subseteq L_\alpha$. De façon équivalente, il existe pour chaque ronde $r \geq R$ au moins un processus $p_i, i \in X_\alpha$ qui obtient un snapshot immédiat sm^r_i dont l'intersection avec X_α est incluse dans L_α. Ceci n'est évidement requis qu'à la condition qu'au moins un processus appartenant à X_α invoque restricted_w_snap() sur l'objet $IS[r]$. La définition suivante formalise la restriction $PR_{\Omega^z_{x,q}}$:

$$\begin{array}{rcl} PR_{\Omega^z_{x,q}} & \equiv & \exists X_1, \ldots, X_q \ : \ (\forall \alpha, \beta : X_\alpha \cap X_\beta = \emptyset) \wedge (x \leq |X_1 \cup \ldots \cup X_q|), \\ & & \exists L_1, \ldots, L_q \ : \ \forall i : (|L_\alpha| \leq z) \wedge (L_\alpha \subseteq X_\alpha), \\ & & \exists R \text{ tels que } \ \forall r : R \leq r, \forall \alpha : 1 \leq \alpha \leq q \\ & & \qquad (\exists i \in X_\alpha : sm^r_i \neq \emptyset) \Rightarrow \big(\exists j(r) \in X_\alpha : (sm^r_{j(r)} \cap X_\alpha) \subseteq L_\alpha\big) \end{array}$$

Il est aisé de vérifier que pour $x = n$ et $q = 1$, $PR_{\Omega^z_{x,q}}$ devient PR_{Ω^z}. De même, lorsque $q = 1$ et $z = 1$, on retrouve la restriction $PR_{\diamond\mathcal{S}_x}$.

4.5.2.2 $\mathcal{SM}_{n,n-1}[\Omega^z_{x,q}]$ *vs.* $IRIS(PR_{\Omega^z_{x,q}})$

Construction de $IRIS(PR_{\Omega^z_{x,q}})$ dans $\mathcal{SM}_{n,n-1}[\Omega^z_{x,q}]$ La construction ne présente pas de difficultés particulières. L'algorithme décrit dans la Figure 4.19 est quasiment identique à celui utilisé pour simuler le modèle $IRIS(PR_{\diamond\mathcal{S}_x})$ dans $\mathcal{SM}_{n,n-1}[\diamond\mathcal{S}_x]$ (voir Figure 4.10). Le prédicat de la ligne 3 autorise un processus p_i à écrire dans l'objet $R[r]$ lorsque l'une des deux conditions suivantes est satisfaite :

- i est l'une des identités des processus que p_i considère comme leaders (i.e., $i \in$ LEADER$_i$) ;
- Parmi les processus considérés comme leaders par p_i, il existe un processus qui a précédemment écrit dans l'objet $R[r]$ (i.e., $m_i \cap$ LEADER$_i \neq \emptyset$).

La terminaison des appels $IS[r]$.restricted_w_snap() repose sue le fait que les ensembles LEADER$_i$ contiennent toujours au bout d'un certain temps l'identité d'un processus correct. D'autre part, dans chaque groupe X_α, les processus « leaders » sont à partir d'une certaine ronde toujours les premiers à écrire dans l'objet $R[r]$.

Lemme 4.7 *L'algorithme décrit dans la Figure 4.19 simule le modèle $IRIS(PR_{\Omega^z_{x,q}})$ dans le modèle $\mathcal{SM}_{n,n-1}[\Omega^z_{x,q}]$.*

Démonstration La construction suit le patron présenté au début du chapitre. Il suffit donc de montrer que les appels $IS[r]$.restricted_w_snap() terminent et que la propriété $PR_{\Omega^z_{x,q}}$ est satisfaite dans toute exécution infinie.

```
operation IS[r].restricted_w_snap(< i, v_i >) :
(1) repeat  m_i ← R[r].snap() ;
(2)         ld_i ← LEADER_i
(3) until ({j :< j, − >∈ m_i} ∩ ld_i ≠ ∅) ∨ (i ∈ ld_i) endrepeat ;
(4) sm_i ← R[r].write_snap(< i, v_i >) ;
(5) return (sm_i)
```

FIG. 4.19 – De $\mathcal{SM}_{n,n-1}[\Omega^z_{x,q}]$ vers $IRIS(PR_{\Omega^z_{x,q}})$ (code pour p_i)

La démonstration de la terminaison est semblable à la démonstration de la terminaison de la construction 4.10. Elle repose sur les deux faits suivants :

1. Soit p_i un processus correct tel que, à partir d'un certain temps, on a toujours $i \in \text{LEADER}_i$. Le prédicat $i \in ld_i$ est donc à partir d'un certain temps toujours vérifié. $\forall r$, p_i ne peut être bloqué indéfiniment dans la boucle **repeat**.
2. Soit p_j un processus correct. Par définition de la classe $\Omega^z_{x,q}$, il existe un processus correct p_ℓ et instant à partir duquel on a toujours $\ell \in \text{LEADER}_j$ et $\ell \in \text{LEADER}_\ell$.

Soient $X_1, \ldots, X_q$ les groupes et $L_1, \ldots, L_q$ les ensembles de « leaders » qui, dans l'exécution infinie considérée, satisfont la définition de la classe $\Omega^z_{x,q}$. Nous montrons que $PR_{\Omega^z_{x,q}}$ est satisfaite pour ces ensembles à partir d'une certaine ronde R. Pour chaque groupe X_α, la sortie du détecteur sous-jacent ne change plus à partir d'un certain τ et est alors toujours égale à L_α. Soit R la première ronde qui démarre après τ.

Nous considérons une ronde $r \geq R$ et un groupe X_α tel qu'il existe $i \in X_\alpha : sm^r_i \neq \emptyset$. Parmi les snapshots immédiats sm^r retournés par les processus appartenant à X_α, nous notons $smin^r_{|X_\alpha}$ le plus petit. Il faut montrer que $smin^r_{|X_\alpha} \cap X_\alpha \subseteq L_\alpha$.

Soit p_j tel que $j \in X_\alpha$ et $j \notin L_\alpha$. Lorsque p_j sort de la boucle **repeat**, $ld_j = L_\alpha$ et donc seule la seconde clause du prédicat de la ligne 3 est satisfaite, c'est à dire $m_j \cap ld_j = m_j \cap L_\alpha \neq \emptyset$. Nous en déduisons que d'après les propriétés de l'objet $R[r]$ (prop. 4.2) il existe $k \in L_\alpha, k \neq j$ tel que $sm^r_k \subsetneq sm^r_j$. D'où $j \in X_\alpha - L_\alpha \Rightarrow j \notin smin^r(X_\alpha)$, c'est à dire $smin^r_{|X_\alpha} \cap X_\alpha \subseteq L_\alpha$. $\square_{Lemme\ 4.7}$

Extraction de $\Omega^z_{x,q}$ dans $IRIS(PR_{\Omega^z_{x,q}})$ La construction décrite dans la Figure 4.21 repose sur le « principe de la roue » introduit dans le chapitre 3, paragraphe 3.4.

Une configuration s décrit un état possible du détecteur de la classe $\Omega^z_{x,q}$ en spécifiant la répartition des processus en groupes $X_1, \ldots, X_q$ ainsi que les ensembles leaders $L_1, \ldots, L_q$ associés. Dans une configuration $s = \big((X_1, L_1), \ldots, (X_q, L_q)\big)$ *valide*, les X_α sont deux à deux disjoints et le cardinal de leur union est égal ou supérieur à x. Chaque couple (X_α, L_α) est telle que $L_\alpha \subseteq X_\alpha$ et $1 \leq |L_\alpha| \leq z$. La sortie LEADER_i du détecteur se déduit de la façon suivante : (1) $\text{LEADER}_i = L_\alpha$ si $\exists \alpha$ tel que $i \in X_\alpha$ et (2) $\text{LEADER}_i = \{i\}$ sinon. Le problème consiste à identifier une configuration *correcte*, c'est à dire telle que $\forall \alpha : (X_\alpha \cap Correct \neq \emptyset) \Rightarrow (L_\alpha \cap Correct \neq \emptyset)$.

Soit donc $\mathcal{S}$ une séquence qui contient toutes les configurations valides possibles. Nous notons nb_S le nombre d'éléments de $\mathcal{S}$. Ces éléments sont indexés entre 0 et $nb\_S - 1$. $\mathcal{S}[k]$ désigne le $k$ième élément de la séquence. La séquence $\mathcal{S}$ est vue comme un anneau logique. Les processus se déplacent dans le même sens sur l'anneau. La position de chaque processus p_i sur l'anneau correspond à une configuration de $\mathcal{S}$. Le déplacement de p_i est régi par les deux règles suivantes :

1. p_i détecte que la configuration qui correspond à sa position courante n'est pas correcte. Il avance alors d'une position sur l'anneau, c'est à dire qu'il examine la configuration suivante.
2. p_i observe un autre processus p_j qui le précède sur l'anneau (i.e., p_j est situé devant lui sur l'anneau ou a effectué plus de tours que p_i). p_i rattrape alors p_j, i.e., la nouvelle position de p_i est celle de p_j.

L'algorithme doit converger vers une position correct. C'est à dire que les processus arrêtent au bout d'un certain temps leur course sur l'anneau et stoppent à la même position qui correspond à une configuration correcte. Le schéma algorithmique de la « roue » nécessite les deux abstractions suivantes pour fonctionner correctement.

- Communication fiable entre processus corrects : try_commit()
  Pour garantir que les processus corrects s'arrêtent sur la même position, tout déplacement d'un processus correct doit être connu des autres processus. La simulation générale (paragraphe 4.4.1, Figure 4.15) émule les primitives write()/snap() dans le modèle $IRIS(PR_{\mathcal{C}})$. La partie du code correspondante est factorisée dans la Figure 4.20.
  La position des processus peut être représentée sous la forme d'un vecteur V. $V[i] = \alpha.nb\_S + \beta$ signifie que p_i a effectué α tours de l'anneau et est maintenant à la position β, c'est à dire qu'il examine la configuration $s = \mathcal{S}[\beta]$. p_i maintient à jour deux vecteurs C_i et T_i en appelant try_commit(C_i, T_i). Lorsque p_i souhaite annoncer sa nouvelle position α, il exécute $T_i[i] \leftarrow \alpha$. Cette position est connue de tous lorsque $C_i[i] = \alpha$. Ainsi, les vecteurs C représente les positions respectives des processus connues de tous, tandis que le vecteur T_i est l'estimation de p_i des positions courantes des autres processus.
- Détecter les configurations corrects : get_smin()
  Le mécanisme de détection s'appuie sur la propriété $PR_{\Omega^z_{x,q}}$. Une configuration s spécifie pour p_i un ensemble X_i et L_i. p_i essaie d'identifier à l'aide de get_smin() le plus petit snapshots $smin_i$ parmi les snapshots obtenus par les processus de X_i pour un objet $IS[r]$. Si $smin_i \neq \emptyset$ et $smin_i \cap X_i \subseteq L_i$ alors s est du point de vue de p_i une configuration correcte. Ce mécanisme simple garantit que (1) tout configuration incorrecte est inéluctablement détectée par au moins un processus correct et (2) lorsque la propriété $PR_{\Omega^z_{x,q}}$ est satisfaite, il existe au moins une configuration correcte qui n'est jamais considérée comme incorrecte.

Émulation de write()/snap() dans $IRIS(PR_{\Omega^z_{x,q}})$ L'algorithme décrit dans la Figure 4.20 reprend le code de l'émulation de la paire d'opération write()/snap() de la simulation générale. La simulation maintient à jour sur chaque processus p_i deux vecteurs C_i et T_i.

C_i contient à tout instant un snapshot valide de la mémoire simulée. Lorsque write(v) p_i souhaite mettre à jour la mémoire, il modifie en l'incrémentant l'entrée i du vecteur T_i.

```
operation try_commit(C_i, T_i)
(1) sm_i ← IS[r_i].restricted_w_snap(T_i); r_i ← r_i + 1;
(2) M_i  ← max_cw{T : T ∈ sm_i};
(3) sm_i ← IS[r_i].restricted_w_snap(M_i); r_i ← r_i + 1;
(4) T_i  ← max_cw{M : M ∈ sm_i};
(5) C_i  ← min_cw{M : M ∈ sm_i};
(6) return(C_i, T_i)
```

FIG. 4.20 – try_commit(), code pour p_i

try_commit() prend en paramètre deux vecteurs d'entiers C et T. Le vecteur C est accessible en lecture seule. Seule l'entrée i du vecteur T_i est modifiable. La seule modification autorisée est le changement de la valeur courante par un entier plus grand. p_i effectue une telle modification lorsqu'il souhaite écrire une nouvelle valeur dans la mémoire partagée.

Nous considérons une exécution dans laquelle chaque processus p_i qui ne subit pas de crash effectue une suite infinie d'appel try_commit(C_i, T_i). Nous notons C_i^ρ et T_i^ρ les vecteurs retournés par le ρième appel try_commit(C_i, T_i) de p_i. Les propriétés énoncées et démontrées dans la preuve de la simulation générale (Lemmes 4.3, 4.5 et 4.6) de la primitive sont résumées dans la définition suivante.

Définition 4.3 (Spécification de try_commit())

1. $\forall \rho, \rho', \forall i, j : (C_i^\rho \leq C_i^{\rho'}) \vee (C_j^{\rho'} \leq C_i^\rho)$
2. *Soit p_i un processus correct. Si p_i effectue un appel try_commit(C_i, T_i) tel que $T_i[i] = \alpha$ alors* $(\exists \rho' : \forall \rho \geq \rho', \forall j : C_j^{\rho'}[i] \geq \alpha)$
3. $\forall$ *try_commit(C_i, T_i) appel effectué par $p_i : T_i[i] \leq \alpha$ alors* $\forall j, \forall \rho : C_j[i] \leq \alpha$.

Implémentation d'un détecteur de la classe $\Omega^z_{x,q}$ dans le modèle $IRIS(PR_{\Omega^z_{x,q}})$
L'algorithme (Figure 4.21) consiste en une boucle infinie. Dans le corps de la boucle, p_i lit la plus grande position atteinte par les processus (ligne 02). Cette position définit la configuration s_i qui va être examinée par p_i (ligne 03) ainsi que la valeur courante de la sortie du détecteur (lignes 04-06). Ensuite, p_i vérifie si sa configuration courante est correcte en identifiant le plus petit snapshot $smin^r_{|X_i}$ obtenu par les processus de X_i pour l'un des objet $IS[r]$. Si il ne parvient pas à identifier un tel ensemble ($smin_i = \emptyset$) ou si cette ensemble n'est pas inclus dans LEADER$_i$, p_i examine la configuration suivante, après l'avoir annoncé aux autres processus (ligne 09).

Dans la preuve, nous considérons une exécution infinie e dans le modèle $IRIS(PR_{\Omega^z_{x,q}})$. Il existe dans e une ronde R, et une suite de couples d'ensembles $(X_1, L_1), \ldots, (X_q, L_q)$ qui satisfont la propriété $PR_{\Omega^z_{x,q}}$. Une phase ρ correspond à l'exécution d'une itération de la boucle **repeat**. Nous notons var_i^ρ la valeur de la variable locale var_i à l'issue de

```
init T_i[1..n] ← [0,...,0]; C_i[1..n] ← [0,...,0]; r_i ← 1; LEADER_i ← {i}

(01) repeat < C_i, T_i > ← try_commit(C_i, T_i); r_i ← r_i + 2;
(02)        mx_i ← max{C_i[j] : j ∈ {1,...,n}};
(03)        s_i ← S[mx_i mod nb_S];
(04)        if ∃(L, X) ∈ s_i such that i ∈ X then X_i ← X; LEADER_i ← L
(05)                                          else X_i ← {i}; LEADER_i ← {i}
(06)        endif;
(07)        smin_i ← get_smin(X_i); r_i ← r_i + (z + 1);
(08)        if (smin_i = ∅) ∨ ((smin_i ∩ X_i) ⊄ LEADER_i)
(09)                                          then T_i[i] ← min(T[i] + 1, mx_i + 1)
(10)        endif
(11) until false endrepeat
```

FIG. 4.21 – Extraction d'un détecteur $\Omega^z_{x,q}$ dans $IRIS(PR_{\Omega^z_{x,q}})$ (code pour p_i)

la phase ρ. Chaque phase s'étale sur $z+3$ rondes car les primitives try_commit() et get_smin() utilisent respectivement 2 et $z+1$ objets $IS[]$ consécutifs. L'exécution de la ρième phase consomme les objets $IS[(\rho-1)(z+3)+1]$, ..., $IS[\rho(z+3)]$.

Lemme 4.8 *$\exists M$ tel que $\forall \rho, \forall i : \max\{C_i^\rho[j] : j \in \{1,\ldots,n\}\} = M$.*

Démonstration Les vecteurs C_i sont mis à jour par l'intermédiaire de la primitive try_commit(). Les valeurs successives de ces vecteur sont donc ordonnées. Soit $(c^1, c^2, \ldots)$ les valeurs successives des vecteurs C_i rangées par ordre croissant et $(m^1, m^2, \ldots)$ la suite des maximums de ces vecteurs, i.e., $m^i = \max\{c^i[j] : j \in \{1,\ldots,n\}\}$. Cette suite d'entiers est croissante et ne contient pas de « trou », c'est à dire que $\forall i : m^{i+1} = m^i + 1$ (ligne 09).

Supposons par contradiction que la suite (m^r) ne soit pas bornée. Remarquons que l'entier m apparaît dans cette suit si et seulement si il existe un processus p_i qui effectue $T_i[i] \leftarrow m$. Soit ℓ l'index tel que $\mathcal{S}[\ell] = \big((X_1, L_1), \ldots, (X_q, L_q)\big)$. Le nombre de processus est fini : il existe donc un processus p_i et une infinité de phases au cours desquelles p_i effectue $T_i[i] \leftarrow m$ avec m tel que $m \mod nb\_S = \ell$. Examinons le code exécuté par p_i lors d'une de ces phases ρ qui démarre lors d'un ronde $\geq R$ (c'est à dire que les index des objets $IS[r]$ utilisés lors de cette phase sont supérieurs à R). Le snapshot $smin_i$ retourné vérifie $smin_i \cap X_i \subseteq L_i$. Ceci est trivialement vrai si $X_i = L_i = \{i\}$. Sinon ceci vient du fait que $PR_{\Omega^z_{x,q}}$ est satisfaite à partir de la ronde R. On en déduit que lors de la phase ρ le prédicat de la ligne 08 n'est pas vérifié. Par conséquent, p_i ne modifie pas $T_i[i]$ lors de la phase ρ : une contradiction.

Nous avons montré que la suite d'entiers (m^r) est croissante et bornée. Par conséquent elle converge vers un entier M. $\square_{Lemme\ 4.8}$

Lemme 4.9 *L'algorithme décrit dans la Figure 4.21 construit un détecteur de défaillances de la classe $\Omega^z_{x,q}$ dans le modèle $IRIS(PR_{\Omega^z_{x,q}})$.*

Démonstration D'après le Lemme 4.8, il existe une phase ρ tel que $\forall \rho' \geq \rho, \forall i : m_i^{\rho'} = M$. Soit $\ell = M \mod nb\_S$. Il reste à montrer que la configuration $s = \mathcal{S}[\ell]$ est correcte.

Soit p_i tel que p_i correct et $i \in X_\alpha$. $smin_i$ vérifie $smin_i \cap X_\alpha \subseteq L_\alpha$. D'autre part, on sait que les sm_i^r contiennent uniquement des identités de processus corrects à partir d'un certaine ronde (Propriété 4.1). D'où $smin_i \subseteq Correct$ et par suite $L_\alpha \cap Correct \neq \emptyset$.

$\square_{Lemme\ 4.9}$

4.5.2.3 Borne pour le (n,k)-accord dans $\mathcal{SM}_{n,n-1}[\Omega^z_{x,q}]$

Dans ce paragraphe, nous établissons une borne sur la calculabilité du (n,k)-accord dans le modèle $\mathcal{SM}_{n,n-1}[\Omega^z_{x,q}]$. Nous avons montré dans la partie précédente que les modèles $\mathcal{SM}_{n,n-1}[\Omega^z_{x,q}]$ et $IRIS(PR_{\Omega^z_{x,q}})$ sont équivalents. C'est à dire qu'il existe une simulation du modèle $IRIS(PR_{\Omega^z_{x,q}})$ dans le modèle $\mathcal{SM}_{n,n-1}[\emptyset]$ $[\Omega^z_{x,q}\ ]$ (Lemme 4.7) et réciproquement il est possible d'extraire un détecteur de la classe $\Omega^z_{x,q}$ dans le modèle $IRIS(PR_{\Omega^z_{x,q}})$ (Lemme 4.9). De plus, le problème auquel on s'intéresse ici est un problème d'accord. Par conséquent, nous nous trouvons dans le cadre d'application du théorème 4.1. Notre étude se concentre donc sur le modèle $IRIS(PR_{\Omega^z_{x,q}})$. Comme indiqué en introduction, nous cherchons à répondre à la question suivante : quel est la plus petite valeur du paramètre k pour laquelle il existe une solution au problème du (n,k)-accord dans le modèle à registre équipé d'un détecteur de défaillances de la classe $\Omega^z_{x,q}$?

Borne inférieure ($k < n - x + qz$) Nous montrons qu'il n'existe pas de solution au problème du (n,k)-accord lorsque $k < n - x + qz$. Nous établissons cette borne par réduction au problème de l'existence d'une solution sans attente pour le $(k+1,k)$-accord dans un système asynchrone composé de $k+1$ processus.

Supposons qu'il existe un algorithme $\mathcal{A}$ qui résout le (n,k)-accord dans le modèle $IRIS(PR_{\Omega^z_{x,q}})$. En analysant un ensemble d'exécutions admissibles dans ce modèle, nous démontrons que $\mathcal{A}$ implique l'existence d'une solution sans attente pour le $(k+1,k)$-accord dans un système asynchrone composé de $k+1$ processus (i.e., dans modèle $\mathcal{SM}_{k+1,k}[\emptyset]$), ce qui est réputé impossible [21, 24, 75, 115].

Proposition 4.8 *Il n'existe pas de solution au problème du (n,k)-accord dans le modèle $IRIS(PR_{\Omega^z_{x,q}})$ si $k < n - x + qz$.*

Démonstration Soit $k < n - x + qz$. Supposons qu'il existe un algorithme $\mathcal{A}$ qui résout le (n,k)-accord dans le modèle $IRIS(PR_{\Omega^z_{x,q}})$. Dans le but d'établir une contradiction, nous considérons une certaine classe d'exécutions E dans le modèle IIS.

Nous divisons l'ensemble des identités en deux ensembles disjoints $L = \{1, \ldots, n - x + qz\}$ et $H = \{n - x + qz + 1, \ldots, n\}$. Un processus p_i dont l'identité i appartient à L est dit processus de *bas niveau*. De même, lorsque $i \in H$ p_i est un processus de *haut niveau*. Une exécution e dans le modèle IIS appartient à la classe E si :

1. $\exists i \in L$ tel que $\forall r : sm_i^r \neq \emptyset$;
2. $\forall \ell \in L, \forall h \in H : (sm_\ell^r \neq \emptyset) \wedge (sm_h^r \neq \emptyset) \Rightarrow sm_\ell^r \subsetneq sm_h^r$.

En d'autres termes, la première condition requiert qu'au moins un processus de bas niveau ne tombe pas en panne. La deuxième indique que les processus de bas niveau qui ne tombe pas en panne sont toujours ordonnancés avant les processus de haut niveau. Ceci implique qu'un processus de bas niveau n'observe jamais de processus de haut niveau. Observons enfin que ces deux conditions impliquent qu'il existe toujours un processus correct parmi les processus de bas niveau.

Nous montrons maintenant que dans toute exécution $e \in E$, la propriété $PR_{\Omega^z_{x,q}}$ est satisfaite. Pour cela, étant donné une exécution $e \in E$, nous définissons une partition de L en $q+1$ ensembles $L_1, \ldots, L_q, O$ qui respectent les conditions suivantes :

- $\forall \alpha, 1 \leq \alpha \leq q : |L_\alpha| = z$;
- L_1 inclut l'identité d'un processus correct.

Une telle partition est réalisable car $|L| = n - x + qz \geq qz$ et dans l'exécution e, il existe au moins un processus de bas niveau correct. Définissons maintenant $X_1, \ldots, X_q$ comme suit :

- $X_1 = L_1 \cup H$;
- $\forall \alpha, 1 \leq \alpha \leq q : X_\alpha = L_\alpha$.

Ces ensembles sont deux à deux disjoints. On vérifie également que $|X| = |\cup X_\alpha| = |H| + \sum |L_\alpha| = x - qz + qz = x$. Il nous reste à montrer que pour chaque ensemble X_α, les processus $\in L_\alpha$ sont ordonnancés avant les autres processus de X_α. Pour $\alpha > 1$, ceci est vrai car par définition $X_\alpha - L_\alpha = \emptyset$. Pour l'ensemble X_1, notons c l'identité d'un processus correct appartenant à L_1. Un tel processus existe par choix de L_1. $\forall r$: $sm^r_c \cap X_1 \subseteq L_1$ car L_1 contient uniquement des processus de bas niveau et ces derniers sont toujours ordonnancés avant les processus de haut niveau de l'ensemble $X_1 - L_1$.

Nous avons exhibé une classe d'exécutions dans le modèle *IIS* qui satisfont la propriété $PR_{\Omega^z_{x,q}}$. En particulier, E contient toutes les exécutions du modèle *IIS* dans lesquelles seuls les processus de bas niveau participent (Les autres processus tombent en panne avant le début de la première ronde). En effet, les deux propriétés qui définissent la classe E n'imposent aucune restriction sur les ordonnancements des processus de bas niveau. Par conséquent, $\mathcal{A}$ résout le (N,k)-accord dans le modèle *IIS* défini pour $N = n - x + qz$ processus. Ceci implique ([24] ou théorème 4.1) une solution dans le modèle $\mathcal{SM}_{N,N-1}[\emptyset]$ pour le $(N, N-1)$-accord ce qui n'existe pas [21, 75, 115]. $\square_{Proposition\ 4.8}$

Solution pour le (n,k)-accord dans $IRIS(PR_{\Omega^z_{x,q}})$ ($k \geq n - x + qz$) Nous démontrons que $PR_{\Omega^z_{x,q}} \Rightarrow PR_{\Omega^{n-x+qz}}$. L'algorithme donné de la Figure 4.17 résout donc le (n,k)-accord lorsque $k \geq n - x + qz$ dans le modèle $IRIS(PR_{\Omega^z_{x,q}})$. Cet algorithme implique l'existence d'une solution dans le modèle $\mathcal{SM}_{n,n-1}[\emptyset]$ (théorème 4.1).

Proposition 4.9 *Il existe une solution au (n,k)-accord dans le modèle $IRIS(PR_{\Omega^z_{x,q}})$ si $k \geq n - x + qz$.*

Démonstration Soit e une exécution infinie dans le modèle $IRIS(PR_{\Omega^z_{x,q}})$. Nous montrons que la propriété $PR_{\Omega^{n-x+qz}}$ est satisfaite dans e. Nous devons montrer qu'il existe un ensemble L et une ronde R tel que (1) $|L| \leq n - x + qz$ et (2) $\forall r \geq R : smin^r \subseteq L$.

Choisissons R comme la ronde à partir de laquelle la propriété $PR_{\Omega^z_{x,q}}$ est satisfaite dans e. Soit $\alpha, 1 \leq \alpha \leq q$ et $i \in X_\alpha - L_\alpha$ (Les X_α et L_α sont des ensembles qui satisfont les conditions de $PR[\Omega^z_{x,q}]$ dans e). Supposons pour établir une contradiction qu'il existe $r \geq R$ tel que $i \in smin^r$. D'où $i \in smin^r \cap X_\alpha$ et par suite $i \in L_\alpha$ car $r \geq R$, ce qui contredit le fait que la propriété $PR_{\Omega^z_{x,q}}$ est satisfaite. Nous en déduisons que $\forall r \geq R : smin^r \subseteq \Pi - \cup_{1 \leq \alpha \leq q}(X_\alpha - L_\alpha)$. Choisissons $L = \Pi - \cup_{1 \leq \alpha \leq q}(X_\alpha - L_\alpha)$. Nous avons alors $|L| \leq n - x + qz$ et $\forall r \geq R : smin^r \subseteq L$: la propriété $PR_{\Omega^{n-x+qz}}$ est donc satisfaite.

Nous avons montré que $PR_{\Omega^z_{x,q}} \Rightarrow PR_{\Omega^{n-x+qz}}$. Or nous savons résoudre le (n,k)-accord dans $IRIS(PR_{\Omega^z})$ lorsque $k \geq z$ (Figure 4.17). Pour $k \geq n - x + qz$, il existe donc une solution au (n,k)-accord dans $IRIS(PR_{\Omega^z_{x,q}})$. $\square_{Proposition\ 4.9}$

Le théorème suivant résume le résultat principale de cette partie.

Théorème 4.2 *Il existe un algorithme qui résout le (n,k)-accord dans le modèle $\mathcal{SM}_{n,n-1}[\Omega^z_{x,q}]$ si et seulement si $k \geq n - x + qz$.*

Démonstration Il existe une extraction d'un détecteur de la classe $\Omega^z_{x,q}$ dans le modèle $IRIS(PR_{\Omega^z_{x,q}})$ (Lemme 4.9). D'autre part, il existe une construction du modèle $IRIS(PR_{\Omega^z_{x,q}})$ dans $\mathcal{SM}_{n,n-1}[\Omega^z_{x,q}]$ (Lemme 4.7). Pour k fixé, il existe donc une solution au problème (n,k)-accord dans $IRIS(PR_{\Omega^z_{x,q}})$ si et seulement si il existe une solution dans $\mathcal{SM}_{n,n-1}[\Omega^z_{x,q}]$ (théorème 4.1). Enfin les propositions 4.8 et 4.9 montrent que ce problème a une solution dans $IRIS(PR_{\Omega^z_{x,q}})$ si et seulement si $k \geq n - x + qz$. $\square_{Théorème\ 4.2}$

Résumé

Dans ce chapitre, nous avons essayé de caractériser la « quantité de synchronie » apportée par les détecteurs de défaillances par l'intermédiaire du modèle snapshot immédiat itéré (*IIS*). L'utilisation de ce modèle est un point crucial dans la démonstrations de certains résultats fondamentaux de la théorie du calcul distribué, notamment le théorème de calculabilité asynchrone ([24, 75, 115]). Étant donné le modèle à registres augmenté avec un détecteur $\mathcal{C}$, le but consiste à définir un modèle qui comporte aussi peu d'exécutions que possible tout en restant équivalent du point de vue de la calculabilité au modèle originel.

Pour $\mathcal{C}$ appartenant aux familles $(\diamond\mathcal{S}_x)_{1 \leq x \leq n}, (\diamond\psi^y)_{0 \leq y \leq n-1}$ et $(\Omega^z)_{1 \leq z \leq n}$, nous définissons un modèle associé $IRIS(PR_\mathcal{C})$ induit par l'ensemble des exécutions du modèle *IIS* qui vérifient une certaine propriété $PR_\mathcal{C}$. Nous montrons que si l'on se restreint à un sous ensemble des problèmes de décisions (les problèmes d'accord), le modèle classique à registres muni de $\mathcal{C}$ et le modèle correspondant $IRIS(PR_\mathcal{C})$ possèdent la même puissance de calcul. Pour illustrer l'approche, nous présentons une borne exacte sur la calculabilité du (n,k)-accord dans le modèle asynchrone muni d'un détecteur $\Omega^z_{x,q}$. La classe $\Omega^z_{x,q}$, dont la définition s'inspire de [71], généralise et unifie les classes Ω^z et $\diamond\mathcal{S}_x$.

La définition du modèle $IRIS(PR_\mathcal{C})$ ne fait pas intervenir explicitement la notion de défaillance : un détecteur $\mathcal{C}$ est caractérisé par une restriction de l'ensemble des exécu-

tions du modèle *IIS*. Ainsi, le modèle $IRIS(PR_{\mathcal{C}})$ capture la capacité d'ordonnancement du détecteur correspondant $\mathcal{C}$. De ce point de vue, un détecteur de défaillances peut être interprété comme un ordonnanceur qui assure inéluctablement certaines propriétés d'(in)équité.

Conclusion

Coordonner des entités réparties demeure l'un des problèmes fondamentaux de la théorie du calcul distribué. La difficulté provient de la nature répartie des systèmes considérés qui génère de nombreuses incertitudes (défaillances, asynchronie, etc.). Dans l'environnement asynchrone, avec défaillances, il est en général impossible de coordonner les entités calculantes (les processus). Afin de mieux comprendre les raisons de ces impossibilités, nous nous intéressons à des tâches de coordination faiblement contrainte ainsi qu'aux circonstances dans lesquelles il existe une solutions à ces tâches. Nous cherchons à classer, sur le plan de la calculabilité les différentes incarnations de la coordination faiblement contrainte. De façon similaire, nous cherchons à comparer les puissances relatives des différentes hypothèses qui ont été introduites pour contrecarrer l'impossibilité de coordonner des processus répartis.

Certaines de ces tâches de coordination faiblement contraintes ont été abstraites sous la forme de problèmes non triviaux dont la formulation est simple. En particulier, nous intéressons aux problèmes suivants : le renommage qui capture l'idée d'allouer à chaque processus une ressource exclusive ; le consensus ensembliste qui requiert de parvenir à une forme de consensus relâché et le test&set ensembliste. Ce dernier problème peut être vu comme un bit test&set dont le fonctionnement est dégradé.

Dans un premier temps, nous classons ces problèmes par rapport à leur difficulté relative. Si l'on dispose d'une solution à l'un de ces problème, est-elle utile pour résoudre un autre problèmes ? Ou au contraire, il n'existe aucun rapport entre eux ? Nous montrons par réductions algorithmiques que, malgré leur nature en apparence différente, il existe une unité forte entre ces problèmes. Nous introduisons également un nouveau problème de coordination faiblement contrainte (la décision de comité) et montrons qu'il capture finement le consensus ensembliste. Ce dernier problème formalise l'idée d'accord sur plusieurs fronts.

Il n'existe pas en général d'algorithmes pour résoudre ces problèmes lorsque l'environnement est complètement asynchrone et les processus tombent en panne. Pour contrecarrer cette impossibilité, la notion de détecteur de défaillances a été introduite. Cette approche abstrait l'hypothèse d'un comportement plus ou moins synchrone du système.

Dans un deuxième temps, nous avons comparé les puissances relatives de détecteurs de défaillances orientés vers la résolution du consensus ensembliste. Pour chaque couple de détecteurs ($\mathcal{C}1$, $\mathcal{C}2$), nous avons donné un algorithme fondé sur $\mathcal{C}1$ qui implémente $\mathcal{C}2$ ou montré l'impossibilité d'une telle transformation. Nous avons également montré

qu'il est possible dans certains cas de combiner $\mathcal{C}1$ et $\mathcal{C}2$ pour obtenir un détecteur $\mathcal{C}3$ strictement plus puissant que $\mathcal{C}1$ ou $\mathcal{C}2$ pris séparément (il est possible de résoudre avec $\mathcal{C}3$ des problèmes qui n'ont pas de solution fondée sur l'utilisation $\mathcal{C}1$ ou $\mathcal{C}2$ seul).

Ce résultat d'addition suggère que l'asynchronie additionnelle abstraite par ces détecteurs n'est pas de même nature. Finalement, nous avons tenté de caractériser la « quantité de synchronie » apportée par les détecteurs de défaillances dans un modèle hautement structuré (le modèle *IIS*). Autrement dit, étant donné la structure des exécutions asynchrones, nous nous demandons quel est l'effet de l'adjonction d'un détecteur sur cette structure. Nous avons montré dans un sens précis qu'augmenter le modèle asynchrone avec un détecteur de défaillances $\mathcal{C}$ induit une restriction sur l'ensemble des exécutions du modèle *IIS*. De ce point vue, un détecteur de défaillances peut être interprété comme un ordonnanceur qui assure inéluctablement certaines propriétés d'(in)équité.

Ces travaux laissent ouvertes un certain nombre de questions. Nous en listons quelques une ci-dessous.

Hiérarchiser la coordination La hiérarchie de Herlihy [69] caractérise partiellement la puissance des objets par rapport à leur capacité à résoudre le consensus parmi c processus. Cependant, cette classification est trop « grossière » pour différencier les objets qui implémentent des problèmes de coordination faiblement contrainte. D'autre part, les relations établies dans le chapitre 2 suggère l'existence d'une certaine unité entre ces objets. Une question importante est donc définir une nouvelle hiérarchie, plus fine, qui engloberait la hiérarchie de Herlihy et expliquerait dans un cadre uni les réductions du chapitre 2. Une telle classification nécessite d'identifier le ou les paramètres pertinent(s) qui caractérise(nt) la difficulté d'un problème de coordination.

Extension à la concurrence non bornée Dans le cadre de ce travail, nous avons fait l'hypothèse d'un nombre fixé n et a priori connu de processus. Cependant, il apparaît dans certains cas que ce paramètre n'est pas pertinent. Par exemple, nous avons montré dans le chapitre 2 qu'il est aussi difficile de construire un objet f_k-renommage pour $n > k$ processus que de construire ce même objet dans un système composé de $n + 1$ processus. Ainsi, un développement naturel du chapitre 2 consiste à étudier les réductions présentées dans le cadre de la concurrence non bornée [85, 56, 3].

Mise en œuvre des détecteurs de défaillances Dans le contexte des détecteurs de défaillances, un aspect important est la question de leur mise en œuvre. En effet, un oracle détecteur de défaillances abstrait des hypothèses de bas niveau sur le systèmes (par exemple : des garanties de temps livraison de messages sur certains canaux de communication ou des temps de réponses de type « *ping* » plus rapides de certains sites). Cependant, la définition d'une classe de détecteurs ne donne aucune indication sur les propriétés concrètes du système. La recherche de la mise en œuvre des détecteurs vise donc à formuler des hypothèses concrètes, notamment de synchronie, sur le comportement du systèmes et à concevoir un algorithme $\mathcal{A}$ qui, lorsque ces hypothèses

sont satisfaites, implémentent un détecteur donné.

Le théorème d'addition montre que du point de vue « détecteurs de défaillances », les informations fournies par les détecteurs $\diamond\psi^y$ et $\diamond\mathcal{S}_x$ ne sont pas de même nature. Cette remarque laissent supposer l'existence d'une famille d'hypothèses, orthogonale à celles introduites pour mettre en œuvre $\diamond\mathcal{S}$ ou Ω, qui autorise la construction de $\diamond\psi^y$. L'existence d'une telle famille ouvrirait la voie vers la conception d'algorithmes implémentant Ω qui bénéficieraient d'une grande couverture d'hypothèses [108].

Étude des détecteurs dans le modèle itéré Le théorème central du chapitre 4 établit une passerelle entre le modèle classique augmenté avec des détecteurs de défaillances et le modèle itéré *IIS*. À partir de cette base, on peut envisager une étude systématique des détecteurs de défaillances. Une possibilité consiste à définir la classe des restrictions du modèle *IIS* qui correspondent qui aux détecteurs de défaillances. La recherche du plus faible détecteur qui permet de résoudre un problème donné pourrait alors en être facilitée du fait de la structure simple du modèle itéré.

Bibliographie

[1] Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., and Shavit, N. Atomic snapshots of shared memory. *J. ACM 40*, 4 (1993), 873–890.

[2] Afek, Y., Dolev, D., Gafni, E., Merritt, M., and Shavit, N. A bounded first-in, first-enabled solution to the ℓ-exclusion problem. *ACM Trans. Program. Lang. Syst. 16*, 3 (1994), 939–953.

[3] Afek, Y., Gafni, E., and Morrison, A. Common2 extended to stacks and unbounded conccurency. *Distributed Computing* (2007).

[4] Afek, Y., Gafni, E., Rajsbaum, S., Raynal, M., and Travers, C. Simultaneous consensus tasks : A tighter characterization of set-consensus. In *Proceedings of the 8th International Conference, (ICDCN'06)* (2006), vol. 4308 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 331–341.

[5] Afek, Y., and Merritt, M. Fast, wait-free $(2k-1)$-renaming. In *PODC '99 : Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1999), ACM Press, pp. 105–112.

[6] Afek, Y., Stupp, G., and Dan, T. Long-lived adaptive collect with applications. In *FOCS'99 : Proceedings of the 40th annual Symposium on Foundations of Computer Sciences* (1999), IEEE Computer Press, pp. 262–272.

[7] Afek, Y., Stupp, G., and Touitou, D. Long-lived and adaptive atomic snasphot and immediate snapshot (extended abstract). In *PODC '00 : Proceedings of the 19th annual ACM symposium on Principles Of Distributed Computing* (2000), ACM Press, pp. 71–80.

[8] Aguilera, M. K., Delporte-Gallet, C., Fauconnier, H., and Toueg, S. On implementing Ω with weak reliability and synchrony assumptions. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing* (2003), ACM Press, pp. 306–314.

[9] Aguilera, M. K., Delporte-Gallet, C., Fauconnier, H., and Toueg, S. Communication-efficient leader election and consensus with limited link synchrony. In *Proceedings of the twenty-third annual symposium on Principles of distributed computing* (2004), ACM Press. To appear.

[10] Almeida, C., and Veríssimo, P. Quasi-synchronism : a step awayfrom the traditional fault-tolerant real-time system models. *Bulletin of the Technical Committee on Operating Systems and Application Environments (TOCS) 7*, 4 (1995), 35–39.

[11] Anderson, J. Composite registers. *Distributed Computing 6*, 3 (April 1993), 141–154.

[12] Anderson, J. H., and Moir, M. Using local-spin k-exclusion algorithms to improve wait-free object implementation. *Distributed Computing 11*, 1 (1997), 1–20.

[13] Arévalo, S., Fernández, A., and Larrea, M. Optimal implementation of the weakest failure detector for solving consensus. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)* (Nürnberg, Germany, 2000), IEEE Computer Society Press.

[14] Attiya, H., Bar-Noy, A., and Dolev, D. Sharing memory robustly in message-passing systems. *J. ACM 42*, 1 (1995), 124–142.

[15] Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., and Reischuk, R. Renaming in an asynchronous environment. *J. ACM 37*, 3 (1990), 524–548.

[16] Attiya, H., and Fouren, A. Polynomial and adaptive long-lived $(2k-1)$-renaming (extended abstract). In *Proceedings of the 14th International Symposium on DIstributed Computing (DISC'00)* (2000), vol. 1914 of *LNCS*, Springer, pp. 149–163.

[17] Attiya, H., and Fouren, A. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM Journal on Computing 31*, 2 (2001), 642–664.

[18] Attiya, H., and Rachman, O. Atomic snapshots in $o(n \log n)$ operations. *SIAM Journal on Computing 27*, 2 (1998), 319–340.

[19] Attiya, H., and Rajsbaum, S. The combinatorial structure of wait-free solvable tasks. *SIAM Journal on Computing 31*, 4 (2002), 1286–1313.

[20] Bar-Noy, A., and Dolev, D. A partial equivalence between shared-memory and message-passing in and asynchronous fail-stop distributed environment. *Mathematical Systems Theory 26*, 1 (1993), 21–39.

[21] Borowsky, E., and Gafni, E. Generalized flp impossibility result for t-resilient asynchronous computations. In *STOC '93 : Proceedings of the twenty-fifth annual ACM symposium on Theory of computing* (New York, NY, USA, 1993), ACM Press, pp. 91–100.

[22] Borowsky, E., and Gafni, E. Immediate atomic snapshots and fast renaming. In *PODC '93 : Proceedings of the twelfth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1993), ACM Press, pp. 41–51.

[23] Borowsky, E., and Gafni, E. A simple algorithmically reasoned characterization of wait-free computations. Tech. Rep. 960020, UCLA, Computer science department, 1996.

[24] Borowsky, E., and Gafni, E. A simple algorithmically reasoned characterization of wait-free computation (extended abstract). In *PODC '97 : Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1997), ACM Press, pp. 189–198.

[25] Borowsky, E., Gafni, E., Lynch, N. A., and Rajsbaum, S. The BG distributed simulation algorithm. *Distributed Computing 14*, 3 (2001), 127–146.

[26] Burns, J. E., and Peterson, G. L. The ambiguity of choosing. In *PODC '89 : Proceedings of the eighth annual ACM Symposium on Principles of distributed computing* (New York, NY, USA, 1989), ACM Press, pp. 145–157.

[27] Chandra, T. D., Hadzilacos, V., Jayanti, P., and Toueg, S. Generalized irreducibility of consensus and the equivalence of t-resilient and wait-free implementations of consensus. *SIAM J. Comput. 34*, 2 (2004), 333–357.

[28] Chandra, T. D., Hadzilacos, V., and Toueg, S. The weakest failure detector for solving consensus. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing (PODC'92)* (Vancouver, BC, Canada, 1992), M. Herlihy, Ed., ACM Press, pp. 147–158.

[29] Chandra, T. D., Hadzilacos, V., and Toueg, S. The weakest failure detector for solving consensus. *J. ACM 43*, 4 (1996), 685–722.

[30] Chandra, T. D., and Toueg, S. Unreliable failure detectors for reliable distributed systems. *J. ACM 43*, 2 (1996), 225–267.

[31] Chaudhuri, S. Agreement is harder than consensus : set consensus problems in totally asynchronous systems. In *PODC '90 : Proceedings of the ninth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1990), ACM Press, pp. 311–324.

[32] Chaudhuri, S. More choices allow more faults : set consensus problems in totally asynchronous systems. *Inf. Comput. 105*, 1 (1993), 132–158.

[33] Chaudhuri, S., Herlihy, M., Lynch, N. A., and Tuttle, M. R. Tight bounds for k-set agreement. *J. ACM 47*, 5 (2000), 912–943.

[34] Chen, W., Zhang, J., Chen, Y., and Liu, X. Partition approach to failure detectors for k-set agreement. Tech. rep., Microsoft Research Asia, 2007.

[35] Chen, W., Zhang, J., Chen, Y., and Liu, X. Weakening failure detectors for k-set agreement via the partition approach. In *DISC'07 : Proceedings of the 21st International Symposium on Distributed Computing* (September 2007). to appear.

[36] Christian, F. Synchronous and asynchronous group communication. *Communications of the ACM 39*, 4 (1996), 88–97.

[37] Chu, F. Reducing Ω to $\diamond\mathcal{W}$. *Information Processing Letters 67*, 6 (1998), 289–293.

[38] Cook, S. A. The complexity of theorem-proving procedures. In *STOC '71 : Proceedings of the third annual ACM symposium on Theory of computing* (New York, NY, USA, 1971), ACM Press, pp. 151–158.

[39] Cornejo, A., Rajsbaum, S., Raynal, M., and Travers, C. Failure detectors as schedulers (an algorithmically-reasoned characterization). Tech. Rep. PI-1838, IRISA, Université de Rennes 1, France, March 2007. (Also Brief announcement, PODC 2007, pp. 308-309).

[40] Cristian, F., and Fetzer, C. The timed asynchronous distributed system model. *IEEE Trans. Parallel Distrib. Syst. 10*, 6 (1999), 642–657.

[41] Delporte-Gallet, C., Fauconnier, H., and Guerraoui, R. (almost) all objects are universal in message passing systems. In *Proceedings of the 19th Symposium on Distributed Computing (DISC'05)* (2005), vol. 3724 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 184–198.

[42] Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Hadzilacos, V., Kouznetsov, P., and Toueg, S. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *PODC '04 : Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 2004), ACM Press, pp. 338–346.

[43] Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., and Kouznetsov, P. Mutual exclusion in asynchronous systems with failure detectors. *J. Parallel Distrib. Comput. 65*, 4 (2005), 492–505.

[44] Dolev, D., Dwork, C., and Stockmeyer, L. On the minimal synchronism needed for distributed consensus. *J. ACM 34*, 1 (January 1987), 77–98.

[45] Dwork, C., Lynch, N., and Stockmeyer, L. Consensus in the presence of partial synchrony. *J. ACM 35*, 2 (1988), 288–323.

[46] Fernández, A., Jiménez, E., and Raynal, M. Electing an eventual leader in an asynchronous shared memory system. In *DSN'07 Proceedings of The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (June 2007), IEEE Computer Society, pp. 399–408.

[47] Fich, F. E., and Ruppert, E. Hundreds of impossibility results for distributed computing. *Distributed Computing 16*, 2-3 (2003), 121–163.

[48] Fischer, M., Lynch, N., and Paterson, M. Impossibility of distributed consensus with one faulty process. *Journal of the ACM 32*, 2 (Apr. 1985), 374–382.

[49] Friedman, R., Raynal, M., and Travers, C. Two abstractions for implementing atomic objects in dynamic systems (also brief announcement, podc 2005, p. 354). In *Proceeedings of the 9th International ConferencePrinciples of Distributed Systems (OPODIS)* (December 2005), vol. 3974 of *Lecture Notes in Computer Science*, Springer, pp. 73–87.

[50] Gafni, E. Round-by-round fault detectors (extended abstract) : unifying synchrony and asynchrony. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing* (1998), ACM Press, pp. 143–152.

[51] Gafni, E. Disc/godel presentation : R/w reductions. DISC'04, 2004. http ://www.cs.ucla.edu/~ eli/eli/godel.ppt.

[52] Gafni, E. Read-write reductions. In *Proceedings of the 8th International Conference, (ICDCN'06)* (2006), vol. 4308 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 349–354.

[53] Gafni, E. Renaming with k-set-consensus : An optimal algorithm into $n+k-1$ slots. In *Proceedings of 10th International Conference on the Principles of Distributed Systems, (OPODIS 2006)* (December 2006), Lecture Notes in Computer Science, Springer Berlin / Heidelberg.

[54] Gafni, E., Guerraoui, R., and Pochon, B. From a static impossibility to an adaptive lower bound : the complexity of early deciding set agreement. In *STOC '05 : Proceedings of the thirty-seventh annual ACM symposium on Theory of computing* (New York, NY, USA, 2005), ACM Press, pp. 714–722.

[55] Gafni, E., and Koutsoupias, E. Three-processor tasks are undecidable. *SIAM Journal on Computing 28*, 3 (1998), 970–983.

[56] Gafni, E., Merritt, M., and Taubenfeld, G. The concurrency hierarchy, and algorithms for unbounded concurrency. In *PODC '01 : Proceedings of the twentieth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 2001), ACM Press, pp. 161–169.

[57] Gafni, E., and Rajsbaum, S. Musical benches. In *Proceedings of the 19th International Conference (DISC 2005)* (September 2005), vol. 3724 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 63–77.

[58] Gafni, E., Rajsbaum, S., Raynal, M., and Travers, C. The committee decision problem. In *Proceedings of the 7th Latin American Symposium on Theoretical Informatics (LATIN 2006)* (March 2006), vol. 3887 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 502–514.

[59] Gafni, E., Raynal, M., and Travers, C. Test&set, adaptive renaming and set agreement : a guided visit to asynchronous computability. In *26th IEEE Symposium on Reliable Distributed Systems (SRDS)* (October 2007), IEEE Computer Society, p. to appear.

[60] Garey, M. R., and Johnson, D. S. *Computers and Intractability : A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[61] Guerraoui, R., Herlihy, M., Kouznetsov, P., Lynch, N., and Newport, C. On the weakest failure detector ever. In *PODC'07 : Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing* (August 2007), ACM, pp. 235–243.

[62] Guerraoui, R., and Kouznetsov, P. On the weakest failure detector for non-blocking atomic commit. In *IFIP 17^{th} World Computer Congress - TC1 Stream / 2^{nd} IFIP International Conference on Theoretical Computer Science (TCS 2002)* (August 2002), pp. 461–473.

[63] Guerraoui, R., and Kouznetsov, P. On failure detectors and type boosters. In *DISC'03 Proceedings of the 17th International Conference on Distributed Computing* (October 2003), vol. 2848 of *Lecture Notes in Computer Science*, Springer, pp. 292–305.

[64] Guerraoui, R., and Raynal, M. A generic framework for indulgent consensus. In *Proceedings of the 23th IEEE International Conference on Distributed Computing Systems (ICDCS'03)* (2003), IEEE Computer Society Press, pp. 88–97.

[65] Guerraoui, R., and Raynal, M. The information structure of indulgent consensus. *IEEE Transactions on Computers 53*, 4 (April 2004), 453–466.

[66] Guerraoui, R., and Schiper, A. Gamma-accurate failuire detectors. In *Proceedings of the 10th Workshop on Distributed Algorithms (WDAG'96)* (1996).

[67] Hadzilacos, V., and Toueg, S. Reliable broadcast and related problems. *Distributed Systems* (1993), 97–145.

[68] Hagit, A., Fouren, A., and Gafni, E. An adaptive collect algorithm with applications. *Distributed Computing 15*, 2 (2002), 87–96.

[69] Herlihy, M. Wait-free synchronization. *ACM Trans. Program. Lang. Syst. 13*, 1 (1991), 124–149.

[70] Herlihy, M., and Aspnes, J. Wait-free data structures in the asynchronous pram. In *SPAA'90 : Proceedings of the 2nd annual ACM Symposium on Parallel Algorithms and Architectures* (1990), ACM press, pp. 340–349.

[71] Herlihy, M., and Penso, L. D. Tight bounds for k-set agreement with limited scope accuracy failure detectors. *Distributed Computing 18*, 2 (November 2005), 157–166.

[72] Herlihy, M., and Rajsbaum, S. The decidability of distributed decision tasks (extended abstract). In *STOC '97 : Proceedings of the twenty-ninth annual ACM symposium on Theory of computing* (New York, NY, USA, 1997), ACM Press, pp. 589–598.

[73] Herlihy, M., and Rajsbaum, S. Algebraic spans. *Mathematical Structures in Computer Science 10*, 4 (2000), 549–573.

[74] Herlihy, M., and Shavit, N. The asynchronous computability theorem for t-resilient tasks. In *STOC* (1993), pp. 111–120.

[75] Herlihy, M., and Shavit, N. The topological structure of asynchronous computability. *J. ACM 46*, 6 (1999), 858–923.

[76] Herlihy, M., and Wing, J. Linearizability : a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (1990), 463–492.

[77] Jayanti, P. Robust wait-free hierarchies. *J. ACM 44*, 4 (1997), 592–614.

[78] Keidar, I., and Shraer, A. Timeliness, failure-detectors, and consensus performance. In *PODC'06 : Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing* (July 2006), ACM, pp. 169–178.

[79] Kouznetsov, P. *Synchronization Using Failure Detectors.* PhD thesis, École Polytechnique Fédéral de Lausanne, 2005.

[80] Lamport, L. On interprocess communication part ii : Algorithms. *Distributed Computing 1*, 2 (June 1986), 86–101.

[81] Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (1998), 133–169.

[82] Lynch, N. *Distributed Algorithms*. Morgan Kaufmann, 1997.

[83] Malkhi, D., Oprea, F., and Zhou, L. Ω meets paxos : Leader election and stability without eventual timely links. In *DISC'05 : Proceedings of 19th International Conference on Distributed Computing* (September 2005), vol. 3724 of *Lecture Notes in Computer Science*, Springer, pp. 199–213.

[84] Matoušek, J. *Using the Borsuk-Ulam Theorem*. Lectures on Topological Methods in Combinatorics and Geometry. Springer, 2003.

[85] Merrit, M., and Taubenfeld, G. Computing with infinitely many processes. In *Proceeddings of the 14th Symposium on Distributed Computing (DISC 2000)* (October 2000), vol. 1914 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 164–178.

[86] Moir, M. Fast long-lived renaming improved and simplified. *Sci. Comput. Programming 30*, 3 (May 1998), 287–308.

[87] Mostéfaoui, A., Mourgaya, É., and Raynal, M. Asynchronous implementation of failure detectors. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN'03)* (2003), IEEE Computer Society Press, pp. 351–360.

[88] Mostéfaoui, A., Mourgaya, E., Raynal, M., and Travers, C. A time-free assumption to implement eventual leadership. *Parallel Processing Letters 16*, 2 (2006), 189–208.

[89] Mostéfaoui, A., Rajsbaum, S., and Raynal, M. Conditions on input vectors for consensus solvability in asynchronous distributed systems. *J. ACM 50*, 6 (2003), 922–954.

[90] Mostéfaoui, A., Rajsbaum, S., and Raynal, M. The combined power of conditions and failure detectors to solve asynchronous set agreement. In *PODC '05 : Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 2005), ACM Press, pp. 179–188.

[91] Mostéfaoui, A., Rajsbaum, S., Raynal, M., and Roy, M. Condition-based consensus solvability : a hierarchy of conditions and efficient protocols. *Distributed Computing 17*, 1 (2004), 1–20.

[92] Mostéfaoui, A., Rajsbaum, S., Raynal, M., and Travers, C. Irreducibility and additivity of set agreement-oriented failure detector classes. In *PODC '06 : Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 2006), ACM Press, pp. 153–162.

[93] Mostéfaoui, A., Rajsbaum, S., Raynal, M., and Travers, C. Research note : From $\diamond\mathcal{W}$ to Ω : A simple bounded quiescent reliable broadcast-based transformation. *J. Parallel Distrib. Comput. 67*, 1 (2007), 125–129.

[94] Mostéfaoui, A., and Raynal, M. Solving consensus using chandra-toueg's unreliable failure detectors : A general quorum-based approach. In *Proceedings of*

the 13th International Symposium on Distributed Computing (London, UK, 1999), Springer-Verlag, pp. 49–63.

[95] Mostéfaoui, A., and Raynal, M. Unreliable failure detectors with limited scope accuracy and an application to consensus. In *Proceedings of the 19th Conference on Foundations of Software Technology and Theoretical Computer Science* (London, UK, 1999), Springer-Verlag, pp. 329–340.

[96] Mostéfaoui, A., and Raynal, M. k-set agreement with limited accuracy failure detectors. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing* (2000), ACM Press, pp. 143–152.

[97] Mostéfaoui, A., Raynal, M., and Travers, C. Crash-resilient time-free eventual leadership. In *Proceedings of the 23rd International Symposium on Reliable Distributed Systems (SRDS)* (October 2004), IEEE Computer Society, pp. 208–217.

[98] Mostéfaoui, A., Raynal, M., and Travers, C. Exploring gafni's reduction land : From Ω^k to wait-free adaptive $(2p - \lceil \frac{p}{k} \rceil)$-renaming via k-set agreement. In *Proceedings of the 20th Symposium on Distributed Computing (DISC'06)* (October 2006), vol. 4167 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 1–15.

[99] Mostéfaoui, A., Raynal, M., and Travers, C. Time-free and timer-based assumptions can be combined to obtain eventual leadership. *IEEE Trans. Parallel Distrib. Syst. 17*, 7 (2006), 656–666.

[100] Mostéfaoui, A., Raynal, M., and Travers, C. From renaming to set agreement. In *Proceeedings of the 14th International Colloquium on Structural Information and Communication Complexity (SIROCCO)* (June 2007), vol. 4474 of *Lecture Notes in Computer Science*, Springer, pp. 66–80.

[101] Mostéfaoui, A., Raynal, M., and Travers, C. Narrowing vs efficiency. Tech. rep., IRISA, Université de Rennes 1, 2007.

[102] Mostéfaoui, A., Raynal, M., Travers, C., Patterson, S., Agrawal, D., and El Abbadi, A. From static distributed systems to dynamic systems. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS)* (October 2005), IEEE Computer Society, pp. 109–118.

[103] Mostéfaoui, A., Raynal, M., and Trédan, G. On the fly estimation of the processes that are alive/crashed in an asynchronous message-passing system. In *Proceedings of the 12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006)* (December 2006), pp. 259–266.

[104] Mostéfaoui, A., Raynal, M., and Tronel, F. From binary consensus to multivalued consensus in asynchronous message-passing systems. *Inf. Process. Lett. 73*, 5-6 (2000), 207–212.

[105] Neiger, G. Set-linearizability. In *PODC '94 : Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1994), ACM Press, p. 396.

[106] Neiger, G. Failure detectors and the wait-free hierarchy (extended abstract). In *PODC '95 : Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1995), ACM Press, pp. 100–109.

[107] Okun, M., and Barak, A. Renaming in message passing systems with byzantine failures. In *DISC'06 Proceedings 20th International Symposium of Distributed Computing* (September 2006), vol. 4167 of *Lecture Notes in Computer Science*, Springer, pp. 16–30.

[108] Powell, D. Failure mode assumptions and assumption coverage. In *Proceedings of the 22nd International Symposium. on Fault-Tolerant Computing (FTCS-22)* (1992), pp. 386–395.

[109] Rachman, O. Anomalies in the wait-free hierarchy. In *WDAG'94 : Proceedings of the 8th International Workshop on Distributed Algorithms* (September 1994), vol. 857 of *Lecture Notes in Computer Science*, Springer, pp. 156–163.

[110] Raïipin Parvédy, P., Raynal, M., and Travers, C. Strongly terminating early-stopping k-set agreement in synchronous systems with general omission failures. In *Proceeedings of the 13th International Colloquium on Structural Information and Communication Complexity (SIROCCO)* (July 2006), vol. 4056 of *Lecture Notes in Computer Science*, Springer, pp. 182–196.

[111] Raipin Parvédy, P., Raynal, M., and Travers, C. Decision optimal early-stopping k-set agreement in synchronous systems prone to send omission failures. In *Proceedings of the 11th IEEE Pacific Rim International Symposium on Dependable Computing* (December 2005), IEEE Computer Society, pp. 23–30.

[112] Raynal, M., and Fernández, A. From an intermittent rotating star to a leader. Pi-1810, IRISA, Université de Rennes 1, 2007.

[113] Raynal, M., and Travers, C. In search of the holy grail : looking for the weakest failure detector for wait-free set agreement. In *Proceedings of 10th International Conference on the Principles of Distributed Systems, (OPODIS 2006)* (December 2006), Lecture Notes in Computer Science, Springer Berlin / Heidelberg, pp. 1–17.

[114] Raynal, M., and Travers, C. Synchronous set agreement : a concise guided tour (including a new algorithm and a list of open problems). In *Proceeedings of the 12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)* (December 2006), IEEE Computer Society, pp. 267–274.

[115] Saks, M., and Zaharoglou, F. Wait-free k-set agreement is impossible : The topology of public knowledge. *SIAM Journal on Computing 29*, 5 (2000), 1449–1483.

[116] Saks, M. E., and Zaharoglou, F. Wait-free k-set agreement is impossible : the topology of public knowledge. In *STOC* (1993), pp. 101–110.

[117] Schiper, A. Early consensus in an asynchronous system with a weak failure detector. *Distrib. Comput. 10*, 3 (1997), 149–157.

[118] Sperner, E. Neuer beweis für die invarianz der dimensionszahl und des gebietes. *Abhandlungen aus dem Mathematischen Seminar der Hamburgischen Universität 6*, 3/4 (September 1928), 265–272.

[119] Tronel, F. *Application des problèmes d'accord à la tolérance aux défaillances dans les systèmes distribués asynchrones.* PhD thesis, Université de Rennes 1, December 2003.

[120] Vitányi, P., and Awerbuch, B. Atomic shared register access by asynchronous hardware. In *FOCS '86 : Proceedings of the 27th IEEE symposium on Foudations of computer sciences* (1986), IEEE, pp. 233–243. Erratum dans FOCS'87.

[121] Welch, J., and Attiya, H. *Distributed computing : fundamentals, simulations and advanced topics.* McGraw-Hill, Inc., Hightstown, NJ, USA, 1998.

[122] Yang, J., Neiger, G., and Gafni, E. Structured derivations of consensus algorithms for failure detectors. In *PODC '98 : Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1998), ACM Press, pp. 297–306.

[123] Zieliński, P. Anti-Ω : The weakest failure detector for set agreement. Tech. Rep. UCAM-TR-694, Computer Laboratory, University of Cambridge, 2007.

[124] Zieliński, P. Automatic classification of eventual failure detectors. In *DISC'07 Proceedings of the 21th International Symposium on Distributed Computing* (2007). To appear.

[125] Zieliński, P. Automatic classification of eventual failure detectors. Tech. Rep. UCAM-TR-693, Computer Laboratory, University of Cambridge, 2007.

Table des figures

Résumé

L'informatique moderne est distribuée. La distribution du calcul résulte parfois d'un besoin applicatif lorsque l'objectif est de connecter des ordinateurs distants. Parfois, elle naît du besoin de tolérer des défaillances. En effet, pour éviter qu'une application ne soit à la merci de la défaillance d'une machine, le calcul est dupliqué sur plusieurs machines.

Au coeur de tout calcul réparti repose une forme de coordination. Le fait même que des ordinateurs aient une tâche commune implique un besoin de se concerter avant d'accomplir certaines tâches. Nous étudions les problèmes de coordination dans le modèle asynchrone, sans hypothèses sur des bornes de vitesse d'exécution des processeurs ou de transmission des messages. Les processus peuvent défaillir à n'importe quel moment.

Le degré de coordination qui peut être atteint en fonction du degré d'incertitude du système est la question principale de cette thèse. Trois formes de coordination sont considérées : l'accord ensembliste, le renommage et le consensus simultané. Dans un premier temps, nous proposons différentes réductions algorithmiques entre ces problèmes, afin de prouver dans quelles conditions une solution à l'un de ces problèmes permet d'obtenir une solution à un autre problème. Nous étudions ensuite des hypothèses nécessaires et suffisantes sur la détection de défaillances permettant de résoudre les problèmes d'accord. Le formalisme utilisé ici est celui des détecteurs de défaillances. Enfin, nous proposons un autre point de vue sur les détecteur de défaillances. Nous caractérisons la puissance de calcul amenée par ces détecteurs par une restriction des exécutions du modèle itéré de Gafni.

Abstract

In an asynchronous distributed system, independent processes run at varying speeds and may even crash ; they communicate through unsynchronized primitives, like sending and receiving messages. To perform shared computation, processes need to coordinate their actions. This is theoretically modeled as solving a coordination task, where processes start with some inputs and have to output values satisfying certain conditions. A fundamental task is consensus, from which it is possible to solve any other coordination task. However, this task is not solvable in asynchronous environments in which process failure may occur.

We study sub-consensus tasks, i.e., coordination tasks that are weaker than consensus. In particular, we focus on renaming which requires processes to rename in a tighter name space, set-consensus which extends consensus by allowing processes to decide on a small number of values and the committee-decision. In the later, processes try to solve simultaneously several instances of the consensus problem and each process is required to decide in at least one instance.

We first explore, through algorithmic reductions, the relationships between these sub-tasks. The second part deals with the use of failure detector to solve set agreement. A failure detector is a distributed oracle that gives hints on failures. We determine the relative computational power provided by various families of failure detectors. The last part demonstrates that failure detectors can be seen as mechanisms that restrict the set of possible executions.